AD-A038 214 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2
LANGUAGE REQUIREMENTS REPORT.(U)
JUL 76 B M BROSGOL, J L FELTY DAHC26-76-C-0006
UNCLASSIFIED IR-179-2 USACSC-AT-76-06 NL

1 OF 2
AD
A038214

20

TECHNICAL DOCUMENTARY REPORT
U. S. ARMY COMPUTER SYSTEMS COMMAND
USACSC-AT-76-06

LANGUAGE REQUIREMENTS REPORT

Authors: B. M. Brosgol
J. L. Felty

January 1977

Prepared for
U. S. ARMY COMPUTER SYSTEMS COMMAND
FORT BELVOIR, VIRGINIA 22060

Prepared by
INTERMETRICS, INC.
701 Concord Ave.
Cambridge, Mass. 02138

Contract No. DAHC26-76-C-0006

DISTRIBUTION STATEMENT
Approved for public release; distribution unlimited.

DISPOSITION INSTRUCTIONS

Destroy this report when no longer needed. Do not return it to the originator.

DISCLAIMER

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: USACSC-AT-76-06	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: LANGUAGE REQUIREMENTS REPORT	5. TYPE OF REPORT & PERIOD COVERED: Interim Report for Period Oct 1975 to April 1976
6. PERFORMING ORG. REPORT NUMBER: IR-179-2	
7. AUTHOR(s): Benjamin M. Brosgol, James L. Felty	8. CONTRACT OR GRANT NUMBER(s): DAHC26-76-C-0006
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Intermetrics, Inc., 701 Concord Ave., Cambridge, Mass. 02138	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 6.27.25A/SX762725DY10/05/001
11. CONTROLLING OFFICE NAME AND ADDRESS: U.S. Army Computer Systems Command (CSCS-AT), Fort Belvoir, Virginia 22060	12. REPORT DATE: July 1976 13. NUMBER OF PAGES: 112
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: 112 pp.	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

18. SUPPLEMENTARY NOTES

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

programming language, high-order language, tactical data systems, management information systems, real-time systems

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

The main purpose of this report is to identify the programming language requirements for the implementation of Army tactical data systems. In addition, language facilities needed for business-oriented management information systems are derived, and are compared with the requirements for tactical systems. The methods employed to identify the needed features include a review of Army responses to questionnaires concerning language requirements; interviews with Army personnel knowledgeable in various tactical systems;

DD FORM 1 JAN 73 1473 EDITION OF NOV 65 IS OBSOLETE

UNCLASSIFIED

Block 20 ABSTRACT (continued)

and examination of system specifications documents and secondary material.

The fundamental result of this study is that, despite some special requirements for programming tactical and MIS systems, there is no inconsistency between the basic language facilities needed for "general purpose" vs. tactical or MIS programming. In particular, the features described in the DoD High Order Language Working Group's "Tinman" document are consistent with the needs of tactical and MIS functions, especially with respect to the language goals of reliability and maintainability.

TECHNICAL DOCUMENTARY REPORT

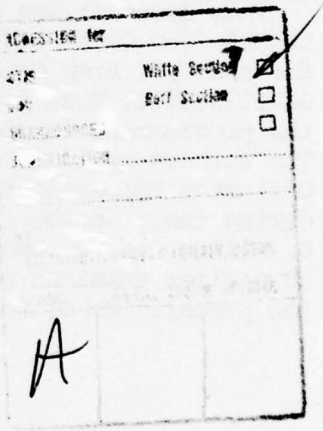
U. S. ARMY COMPUTER SYSTEMS COMMAND

USACSC-AT-76-06

LANGUAGE REQUIREMENTS REPORT

January 1977

Authors: B. M. Brosgol
J. L. Felty



Prepared for

U. S. ARMY COMPUTER SYSTEMS COMMAND

FORT BELVOIR, VIRGINIA 22060

Prepared by

INTERMETRICS, INC.
701 Concord Ave.
Cambridge, Mass. 02138

Contract No. DAHC26-76-C-0006

DA Project SX762725DY10

DISTRIBUTION STATEMENT

Approved for public release; distribution unlimited.

ABSTRACT

The main purpose of this report is to identify the programming language requirements for the implementation of Army tactical data systems. In addition, language facilities needed for business-oriented management information systems are derived, and are compared with the requirements for tactical systems. The methods employed to identify the needed features include a review of Army responses to questionnaires concerning language requirements; interviews with Army personnel knowledgeable in various tactical systems; and examination of system specifications documents and secondary material.

The fundamental result of this study is that, despite some special requirements for programming tactical and MIS systems, there is no inconsistency between the basic language facilities needed for "general purpose" vs. tactical or MIS programming. In particular, the features described in the DoD High Order Language Working Group's "Tinman" document are consistent with the needs of tactical and MIS functions, especially with respect to the language goals of reliability and maintainability.

FOREWORD

This document was prepared under the authority of U.S. Army Contract No. DAHC26-76-C-0006 as CDRL Item A003, and was prepared by Intermetrics, Inc. for the U.S. Army Computer Systems Command. The DA Project Number is SX762725DY10, Task Area is 05, and Work Unit is 001. This study identifies the programming language requirements for Army tactical and MIS applications. The authors would like to acknowledge the guidance of the Army's Contracting Officer's Representatives -- Major Benjamin D. Blood, Jr. and Lt. Brent Price -- during the preparation of this document. We also are grateful to Dr. James S. Miller and Mr. Dennis D. Struble of Intermetrics, who provided comments on earlier versions of this report, and to Mr. John Bates, who supervised the preparation of the manuscript.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

Section I. SCOPE

1. Background.

a. High Costs of Software. One of the most striking trends in data processing over the past several years has been the increase in the ratio of software to hardware costs, particularly for large systems. This is perhaps not surprising: technological advances in hardware have resulted in faster and more powerful machines for which previously unattainable applications become possible. The complexity of such applications, however, demands correspondingly complex programming, and the difficulties of producing such software have contributed to the increase in costs.

b. Role of Language. In choosing a language (or set of languages) in which to produce software systems, an agency such as the Army has the potential to influence significantly the cost of the resulting systems. The language is the programmer's main tool for shaping the final product, and it has a direct effect on how the programmer thinks about solving the problem. A language not well suited to the particular application will make the programming task more difficult and error-prone, and the software more expensive to produce, verify, and maintain. Unfortunately, the language selection process has not often received the priority it deserves. The effect is a proliferation of high-order languages (HOLs) as well as an over-dependence on machine language, resulting in a lack of software portability and increased expenses in programmer training and language maintenance.

c. Common HOL. The problem of HOL proliferation, and the desirability (from the perspective of a large software-procuring agency) of HOL standardization, are not new issues. For example, DoD Directive 4630.6 in January 1967 directed that efforts be made to promote system compatibility and commonality. More recently, a Working Group has been established under the auspices of DDR&E to determine a minimal set of HOLs to be used throughout the Department of Defense. Under sponsorship of this Working Group, several documents have been prepared and circulated describing the characteristics of a common HOL for DoD applications; these documents will be referred to as the "Strawman" [5], "Woodenman" [6], and "Tinman" [7].

d. Purpose of Current Contract. The current contract from the Army to Intermetrics is in conjunction with the Army's participation in the HOL Working Group. Intermetrics' effort under this contract involves two general activities: the examination of the application area of tactical data systems, to determine the language requirements implied by such systems; and the evaluation of a set of candidate HOLs against the requirements established for a common language.

2. Purposes of This Report.

a. Main Purpose. The main purpose of this report is to identify the programming language requirements for the implementation of Army tactical data systems. (The term "implementation" refers to the entire system construction process, comprising design, programming, fielding, testing, and maintenance.) The language requirements fall in two categories: language goals, which must be met in order to satisfy general objectives of tactical systems; and language features, which must be present to facilitate the programming of specific tactical functions. These derived requirements are used as the basis for a comparison with the language characteristics proposed in the Tinman.

b. Secondary Purpose. While the main focus of this report is on tactical data systems, attention is also given to business-oriented management information systems (MIS). Programming language requirements for this application area are derived, the differences and similarities between the requirements for MIS and tactical systems are examined, and a comparison with the Tinman is conducted.

Section II. APPROACH

1. Basic Approach.

This report takes a "function-oriented" (as opposed to "feature-oriented") approach[1] to the derivation of language requirements. Tactical and MIS programs are categorized based on the functions which they carry out, and the implications of these functional requirements, on the programming language which is used to implement the system, are discussed.

2. Sources.

Several types of sources were used in the preparation of this report. Direct information on requirements for Army systems was obtained from a set of Army responses to questionnaires concerning the Strawman and Woodenman, from interviews at USACSC with individuals knowledgeable in the various systems, and from USACSC fact sheets describing the systems. Additional material was derived from Intermetrics' experience with the implementation of systems with similar functional requirements, and from various secondary sources.

---

[1] Examples of "feature-oriented" approaches are the Tinman and SofTech's study of language features [18].

Section III. OVERVIEW OF DOCUMENT

This report is divided into five chapters, with the current chapter providing the background and introductory material. Chapter 2 presents the programming language requirements for tactical data systems and compares these with the characteristics listed in the Tinman. Chapter 3 presents the programming language requirements for management information systems, contrasts these with the requirements for tactical systems, and compares these also with the Tinman. Chapter 4 summarizes the conclusions of this study.

CHAPTER 2

REQUIREMENTS OF TACTICAL DATA SYSTEMS

Section I. AN OVERVIEW OF TACTICAL DATA SYSTEMS

1. Army Tactical Data Systems.

This paragraph discusses the basic features and purposes of several Tactical Data Systems currently under development by USACSC. The main references for the information presented are [30], [31], [32], and [33].

a. TACFIRE. TACFIRE (Tactical Fire Direction System) is an on-line, real-time tactical system to be used in the Army's field artillery units during the 1978-1981 time period.

(1) Purpose. As stated in [30], "The objective of TACFIRE is to increase the effectiveness of field artillery fire support through increased accuracy, better and more rapid use of target information, reduced reaction time and greater efficiency in the determination of fire capabilities and the allocation of fire units to targets. Specifically, TACFIRE is the application of automatic data processing techniques to the seven field artillery functions of technical fire control, tactical fire control, fire planning, artillery target intelligence, artillery survey, meteorological data, and ammunition and fire unit status. The system will also provide the capability for the fire support element, which is part of the tactical operations center, for preliminary target analysis, nuclear target analysis, nuclear fire planning, chemical target analysis, and fallout prediction."

(2) Hardware.

(a) Digital computer. A modular third-generation military computer (AN/GYK-12) will be used, with expansion and self-checking capabilities.

(b) Control console. An Artillery Control Console (ACC) will enable operating personnel to control program functioning. The ACC will be used for entering messages and data into the computer, displaying data and messages, and indicating malfunctions.

(c) Local and remote input/output equipment.

1 Message entry devices. Such devices will be used for sending data to the computer center from remote locations, over radio or wire voice nets. Both Digital Message Devices (DMDs) and Variable Format Message Entry Devices (VFMEDs) will be provided.

2 Display. The artillery battery will have a Battery Display Unit (BDU) for visual display and permanent printout of necessary information.

3 Printers. Medium Speed Printers (MSPs) will be located at each computer center. An Electronic Line Printer (ELP) in each S-280 TACFIRE shelter will provide hard copy printouts of activity.

(d) Digital data storage and retrieval units. An External Auxiliary Removable Media Memory Unit (ARMM) will be used to load computer memory and store data. Mass Core Memory Units (MCMU) will provide the necessary core capability.

(e) Graphical display units. A Digital Plotter Map (DPM) will display the current tactical situation at Battalion, and an Electronic Tactical Display (ETD) will augment the DPM at the Division Artillery Fire Direction Center.

(f) Communication co-ordination. A Digital Data Terminal (DDT) or Remote Data Terminal (RDT) will interconnect the communication nets with the computer, VFMED, BDU, and communications security equipment.

(3) Software.

(a) Executive. This component consists of an Executive Kernel and various Supervisors. The Executive Kernel fields and processes interrupts; performs services such as job creation, resource contention control, memory allocation, timer maintenance, display refreshing, and hardware monitoring; and takes care of I/O control services. The Supervisors attend to file management (core files in the field system), message processing, salvage point recording, system initialization, degraded modes capability, system security, system purge, dumps for debugging, and operator communications.

(b) Application programs. These solve specific TACFIRE problems. Major types of required processing include ballistic computation based on meteorological and target range and location data, and data correlation and conflict resolution.

(c) Utility programs. These include programming aids (which assist in the preparation and debugging of programs), compilers (for TACPOL into AN/GYK-12 machine language), and programs for maintenance and diagnostics (which permit continual monitoring of system performance and rapid detection of faults).

(4) System operation. A simplified diagram of the data paths in a TACFIRE system is given in Figure 1.

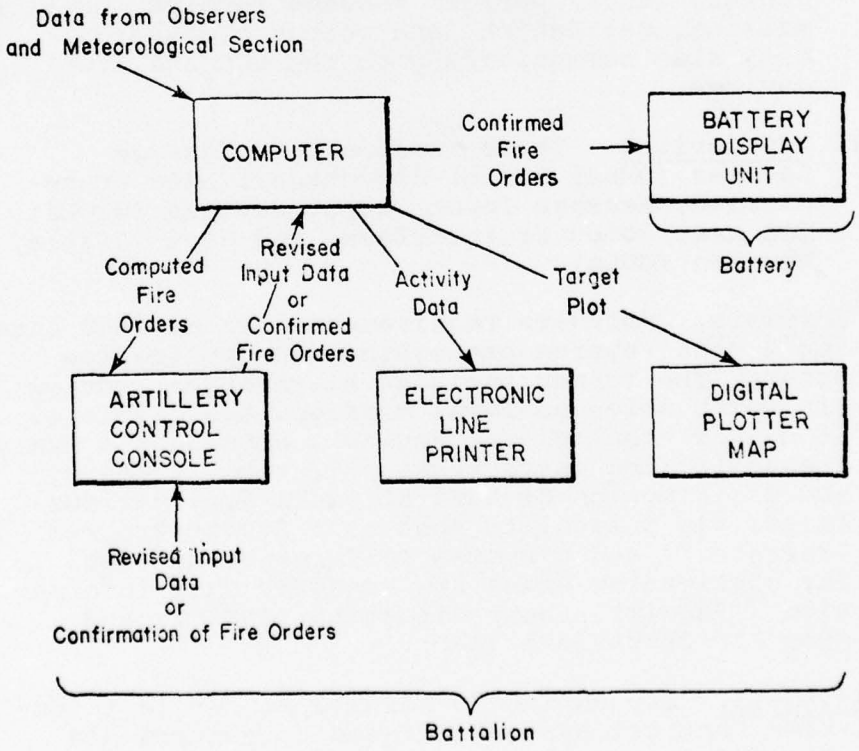


Figure 1. TACFIRE System (simplified)

b. TOS. TOS (Tactical Operations System) will be an on-line, real-time tactical system to be used at a variety of echelons, to assist decision-making in the areas of command, operations, intelligence, and support. The TOS Operable Segment (TOS2) is a militarized tactical data system test bed currently under development for field testing TOS concepts. The discussion below pertains mainly to TOS2 and was derived from [31] and [33, pp. 12 ff.].

(1) Purpose. The basic objective of TOS is to assist the commander and his staff by applying modern automatic data processing techniques to provide more timely, accurate, and complete tactical information.

(2) Hardware.

(a) Computer centers. A Central Computer Center (CCC) performs total data base processing for all data entering the system. Remote Computer Centers (RCCs) perform message receipt functions, editing, validation, and output processing; they also communicate with the CCC and I/O devices.

(b) I/O devices. These include Data Message Devices (DMDs) for bi-directional data transmission, Message Input/Output Devices (MIODs) for user/computer interface, and Group Display Devices (GDDs).

(3) Software. Software requirements are divided into two areas: system processing and application areas. The former includes hierarchical review (in which selected input messages are displayed at higher echelons for review), commander's query (requiring the extraction, collation, summary, and distribution of data to users from various files, via a facility such as a Report Program Generator), and a system performance monitor. The application areas are Friendly Unit Information (FRENSIT), Enemy Situation (ENSIT), and Army Air Operations (AAO).

c. AN/TSQ-73. The AN/TSQ-73 Missile Minder is an on-line, real-time tactical system designed to control and coordinate air defense weapons for the Army Air Defense Command Post.

(1) Purpose. The objective of AN/TSQ-73 is to increase the effectiveness of Army Air Defense Units against the present and near future air threat, through the use of microelectronic technology and advanced maintenance philosophy.

(2) Hardware. An AN/TSQ-73 system comprises a repackaged (for transportability) configuration of the AN/GYK-12 computer, peripheral devices, display consoles, radar data processing equipment, and digital data links. The system operates in a dual-CPU environment with seven 8K memory modules. CPU-1 acts as a master CPU and performs executive functions as well as message and display processing. CPU-2 performs some executive functions but is mainly devoted to tracking: it correlates, associates and updates tracks. All files are core resident.

(3) Software.

(a) Executive. The executive component performs system initialization, system control, control of some peripheral devices, system bookkeeping, and control of time-dependent functions [34, pp. 4 - 5].

(b) Application programs. The functions performed include: air-space surveillance, target tracking, identification, display and data link communications, and on-line confidence testing.

(c) Utility programs. These include: an assembler; simulation programs for use in on-site operator training and equipment exercising; fault detection and isolation software; and support programs such as system tape generation.

2. General Properties of Tactical Data Systems.

a. Scope. This paragraph summarizes the main features of Tactical Data Systems (hereafter abbreviated as "TDS") and establishes a framework for subsequent presentation of the requirements of such systems. At the outset it should be pointed out that the differences between TDS and non-tactical systems are sometimes minor; moreover, there are often important differences between tactical systems themselves. Nonetheless, it is possible to abstract a set of common properties of "typical" TDS. The sources of this information include the survey of Army systems presented in the preceding

paragraph; interviews at USACSC with persons knowledgeable in the various systems; letters and responses from individual Army sites in connection with their data processing requirements as reflected in the Strawman [5] and Woodenman [6] reports; and secondary sources such as Martin [1], Aron [19], and Chapin [20].

b. Types of Tactical Systems. All TDS fall in the category of Real-Time Computer Systems (RTCS). Following Martin [1, p. 4], we define a RTCS as "one which controls an environment by receiving data, processing them and returning the results sufficiently quickly to affect the functioning of the environment at that time". In addition to operating in real-time, most TDS are "on-line": i.e., there is no manual intervention (such as tape mounting) between the time the data are input and the time the results are output. The following sub-paragraphs identify the two basic varieties of RTCS which most TDS exemplify (the terminology is derived from Aron [19]).

(1) Control systems. TDS such as TACFIRE and AN/TSQ-73 exhibit properties of specialized process control systems. General features include mathematically-based control algorithms whose execution times are precisely known, short response times, and input data which may be either periodic (e.g., radar scans) or random (e.g., information forwarded by manual observers). The system may exercise control either directly or by outputting commands to personnel.

(2) Command systems. A command system's main purpose is to supply information rather than control an environment. Inputs are in the form of user queries, the processing involves the analysis of a large data base, and the output is the result of the data base interrogation. Since the performance requirements for such systems are frequently in a state of flux, flexibility is important. An example of a system which exhibits command system properties is the Army's TOS; also, TACFIRE and AN/TSQ-73 combine elements of command with control. In TACFIRE the Fire Support Element functions can be considered command functions. In AN/TSQ-73, air space surveillance and weapons assignment can be considered command functions. The primary difference is that in systems such as TACFIRE and AN/TSQ-73, the mission objectives allow the data to be more quantifiable than the data required for systems such as TOS.

c. Purposes of Tactical Systems. Since command systems are basically Management Information Systems, and since MIS will be discussed in detail in Chapter 3, we will for the remainder of the present chapter focus our attention on TDS which are control systems. The main purposes of such TDS, cited in Chapin [20, p. 787], are:

(1) to collect data from local and remote sources and systems;

(2) to correlate the data in order to gain an accurate view of the tactical situation;

(3) to process the data and derive the appropriate control decision;

(4) to communicate the decision to personnel, or other systems, or to carry out the decision via weapons control.

d. Constraints on Tactical Systems. Many of the distinguishing features of TDS are derived not so much from the functions performed as from the severe constraints under which such systems must operate. Among the more important requirements are the following:

(1) TDS must be able to function in the presence of inaccurate and/or false data inputs, which are caused by such sources as radar "clutter" and radar beacon "garbles".

(2) Short response times are necessary (e.g., 50 to 500 msec) in the presence of a large number of inputs, possibly arriving at unpredictable times and possibly peaking at any time.

(3) TDS must perform reliably, even in the event of operator errors and equipment failures and must be secure from penetration and compromise.

(4) 24 hour/day operation is necessary.

(5) Because of physical size limitations on hardware (caused by either environmental constraints or by economic considerations based on the multiplicity of implemented systems), storage for the software comprising the operational system is sometimes limited.

(6) TDS must communicate with other Army systems and sometimes other service systems.

e. Functional Subdivision of Tactical Systems. TDS typically contain a large variety of programs which perform a wide range of functions and operate under a variety of constraints. To deal with this diversity in a systematic way, we will classify the software comprising a TDS into three categories, corresponding to the function which is carried out. The following subparagraphs summarize the distinguishing characteristics of these three areas; details concerning the functional requirements will be provided in Sections II through IV below.

(1) Application Programs. These are the programs which carry out the data correlation and data processing functions of the TDS. Tracking, weapons assignment, and control are some of the types of processing found in application programs.

(2) Executive Programs. These programs (called "supervisory" in Martin [1]) perform the functions associated with task scheduling and synchronization, input/output, and interrupt handling. The executive software basically serves as a special-purpose real-time operating system.

(3) Utility Programs. The utility or "support" programs perform a variety of ancillary functions which are necessary for the installation or smooth operations of the TDS. Examples of these programs are system construction facilities, system simulators, automated test equipment routines, and software testing aids. Compilers or assemblers for languages used in developing the TDS may be regarded as part of the utility software. Since such programs are not part of the running system, they generally do not have the strict efficiency requirements of the application or executive programs.

Section II. REQUIREMENTS OF TDS APPLICATION PROGRAMS

This section describes the major functions performed by TDS application programs, and discusses the language implications for each function.

1. Tracking.

a. General Functions. Automatic tracking involves three general problems: data prediction, data correlation, and data combination. The solutions to these problems entail high sampling rates and complex correlation techniques, and generally apply techniques from linear estimation theory (e.g., Kalman filtering) to a time-discrete model. Such methods make use of matrix recursion relations to estimate various states of the system.

b. Data Packing. A history of track data must be maintained during processing. Since there are circumstances under which the programmer would require those data to be packed (i.e., core memory constraints) and other circumstances under which non-packed data would be preferred (i.e., requirements for high execution speed), the language should provide a means of specifying the packing with the definition of the data structure.

c. Arithmetic Data Types. The issue of whether to use fixed or floating point arithmetic in the implementation of tracking procedures has been the source of some controversy. As mentioned in [35, p. 15]:

> Considerable effort has been expended to show that most calculations associated with the tracking function can be done in fixed point arithmetic. This is true; however, it implies a considerable amount of preknowledge about the dynamics of the system to be calculated, and necessitates considerable special coding to prevent scaling problems. A floating point arithmetic capability does not eliminate these requirements, but does reduce them considerably while providing an increase in the basic numerical significance of the system at the cost of some additional round off error. Experience and analysis, however, conclusively indicate that the latter inefficiency is more easily tolerated than are the problems associated with fixed point.

Language implications are for floating-point arithmetic to be provided, and, in light of the fact that some TDS hardware may not support floating-point, for a fixed-point facility to be available.

d. Mathematical Routines. Conversion of sensor inputs into a form suitable for mathematical processing is required; e.g., polar to cartesian coordinates. Thus trigonometric and other numerical procedures must be available in the language.

e. Vectors and Matrices. Associated with tracking is the problem of track detection (i.e., the determination of the presence of a track or vehicle at a given point in time within a given space, where such a track or vehicle did not previously exist). Solutions are statistical in nature and involve vector and matrix calculations.

2. Guidance.

Weapons guidance involves the functions of navigation (determining missile position as a function of time), steering (establishing a specified terminal position) and tracking (of both target and missile). Requirements for navigation software range from adaptive filters to dead-reckoning; these in turn place a demand for matrix operators, floating point arithmetic, and matrix parallel processing. Steering functions have similar requirements, with an even greater speed constraint.

3. Threat Evaluation.

This function involves the detection of threat elements and the determination of the individual characteristics of such elements. Various levels of sophistication are possible for the performance of this function, ranging from simply establishing the presence of a threat, to evaluating threat posture and structure, to conducting threat analysis and computing the required time to destruct. Interaction with operating personnel is standard. Processing requirements include sensor data analysis, track prediction, data base handling, statistical computations, and intelligence data processing. The language requirements are basically the same as for tracking.

4. Weapons Assignment.

The weapons assignment function involves threat analysis, resource management, and weapons selection and deployment. Statistical computations, data base handling, and extensive man/machine interfaces are required.

5. Electronic Warfare.

This function includes both Electronic Counter Measures and Electronic Intelligence. Flexibility and modifiability

of the software is important; as stated in [35, p. 20], "The problem ... for the tactical data system is to provide software which will satisfy the existing operational need and be structured to readily permit changes as new operational requirements and new counter move solutions develop". Processing requirements include correlation, spectrum analysis, and filter processing (using Fast Fourier Transform techniques), handling large volumes of data, high speed sorting and searching, signal separation, triangulation, and deghosting.

6. Communications.

a. Use of Assembly Language. As pointed out in recent reports by the Air Force [36, p.11] and the Defense Communications Engineering Center [37, p. 1], tactical systems in the past have relied almost exclusively on assembly language for the programming of communications software. Execution-time efficiency, main memory limitations, and unavailability of a HOL are some of the considerations cited in [36, p. 11] for the selection of assembly language. While some of the functions performed by the communications software require assembly language, a large portion of the processing can be carried out in a HOL with suitable facilities. In the following subparagraphs, we describe the major functions characteristic of communications programs [1, pp. 114 - 116] and indicate the language implications.

b. Data Reception. The program must initiate and control the reception of data from one or more lines with possibly different transmission rates. Executive routines for handling interrupts and scheduling time-critical processes (see paragraph 2.III.3) are necessary, as well as facilities for dealing with untyped data.

c. Message Formatting. This function entails assembling bits into characters, and characters into messages, possibly converting the coding of characters from external code (e.g., Baudot) to internal (e.g., ASCII). An ability to handle bit- and character-string data is needed here.

d. Error Checking. To carry out this function (e.g., via parity checking), the program needs facilities for treating characters as bit-strings and performing operations such as masking.

e. Message Delivery. Process synchronization facilities are required in the executive in order to enable messages to be sent to, and received from, other application programs.

f. Data Formatting and Transmission. These functions are the reversals of b and c above: data to be transmitted must be broken down into bits for sending, and the transmission activity must be initiated and controlled. The required language facilities are the same as for b and c.

7. Display Processing.

a. Man-Machine Interface. A critical function of display processing is to provide the interface to the personnel who will manually insert or retrieve data. There are a variety of approaches possible, with tradeoffs between efficiency of implementation and simplicity of use:

(1) Pre-structured formats. This type of interface is prevalent in existing systems and achieves efficiency of operation at the expense of flexibility and user convenience. In this approach the user fills in blank forms (from a console keyboard) by typing entries according to a fixed set of rules. The language features required to implement such an interface involve primarily the processing of character strings. The kind of processing needed (cited by Shaw [29]) involves: declaration of string variables of different lengths, and operations such as assignment, relations, concatenation, and substring.

(2) Interactive queries. Drawbacks to pre-structured formats have been cited in a study by Larson [38], based on tests undertaken at Modern Army Selected Systems Test Evaluation and Review. The main problems are the requirement for intermediary personnel to assist the users in developing the appropriate formats, and the relative inflexibility of the system with respect to changes in the kinds of information entered or retrieved. An alternative approach is to provide as the man-machine interface a more natural interactive query language. The language facilities required include those necessary to handle pre-structured formats (i.e., character string processing) as well as powerful data definition facilities (to specify the structure of the data base and to deal with the data structures needed for linguistic processing).

b. Picture Generation. A second critical requirement in display processing involves the generation and updating of pictures of two- or three-dimensional configurations. Programs carrying out these functions need facilities for

performing the mathematics of the problem (e.g., rectangular to polar conversions, scaling, translation, rotation); trigonometric routines are heavily used. In addition, the programs must interface with the executive routines which actually control the display device; time-critical scheduling and interrupt handling (for display refreshing), as well as features which allow the specification of memory word contents, are required.

8. TACFIRE-Oriented Functions.

The various field artillery functions performed by TACFIRE may be summarized as follows. The technical fire control function is concerned with ballistics, settings, and damage prediction. Tactical fire control, a somewhat simpler function, deals with the selection of fire units. Fire planning is concerned with the scheduling of fire for future use, and is constrained by storage requirements. Artillery target intelligence accepts inputs from manually controlled fixed-format message entry devices. These inputs include reports on POWs, radars, and enemy shellings (caliber, direction) and are used to help determine target postion. The artillery survey function, which can operate in stand-alone mode, solves formal survey problems (traverse, triangulation, resection) involving friendly artillery. The meteorological data function, used in ballistic computations and fallout prediction, enters and stores data in appropriate tables. The ammunition and fire unit status function is effectively an inventory control, accepting manual inputs from an operator. Future requirements are possible for application programs in the tracking area, with laser inputs. The main processing performed is mathematical in nature; floating point facilities in the language would be useful.

Section III. REQUIREMENTS OF TDS EXECUTIVE PROGRAMS

1. Purposes of Executive Programs.

a. Main Purposes. The main purposes of executive software in TDS are to ensure the efficient and reliable sharing of resources among the tasks that comprise the application programs, and to guarantee the timely processing of the various sensor and manual inputs to the system. The distinctive features of tactical executives, as contrasted to commercial operating systems, lie in the critical efficiency and reliability constraints, the (sometimes) more predictable nature of the application programs, and the highly diverse assortment of I/O equipment to be handled.

b. Resource Management. The view of a tactical executive as a manager of resources allows a useful framework for the discussion of language requirements. Following Madnick and Donovan [2], we distinguish among four types of resources: memory, processors, I/O devices, and information (programs and data). In paragraphs 2 through 5 below, we consider the functional requirements and resulting language implications for the management of these four resources.

2. Memory Management.

a. General Functions. The memory[1] management functions involve the following activities [2, p. 105]:

(1) Keeping track of which locations are in use (or "allocated") and which are free.

(2) Enforcing an allocation policy, i.e., deciding which tasks should be allotted memory space, and how much.

(3) Actually allocating the memory.

(4) Enforcing a memory reclamation policy.

---

[1] The term "memory" here refers to directly accessible (or primary) memory, as opposed to on-line devices like disks, etc.

b. Variety of Memory-Management Strategies. The policies available for memory management vary widely in sophistication and applicability. At the simplest level is "single contiguous allocation", in which memory is divided into two areas: space occupied by the executive, and space allocated to a single running process. Despite the simplicity of this scheme, its inherent inefficiencies (e.g., poor utilization of memory, idling of processor during I/O waits) make it unacceptable in tactical environments. At the other extreme are flexible and general techniques found in modern operating systems like OS/VS-2 for the IBM 370, and MULTICS for the Honeywell 6180. Those systems combine demand-paged memory management (where processes need not be loaded into a contiguous memory area, and where the total space required for a process may exceed the available memory space) and segmented memory management (where an address comprises a segment specification and a relative offset within that segment, and where a segment may be of arbitrary size). Such general schemes allow for considerable flexibility in managing memory, but entail "increased hardware cost, processor overhead for address mapping, the need for additional memory space for tables, and increased complexity in the operating system" [2, p. 181]. As a result, such memory management techniques are generally unsuitable in a tactical executive; a more appropriate technique is described in the next paragraph.

c. Partitioned Allocation. A more reasonable policy for memory management in TDS is "partitioned allocation". In this scheme, memory is divided into "partitions", each housing a separate process. The number of partitions gives the "degree of multiprogramming" supported: in general, the larger this number, the greater the probability that the processor will not be idle during I/O wait.[1] Since tactical systems typically consist of a fixed number of processes, the management of memory partitions can frequently be done "statically", with each process or shared data region allocated (at compile time) to a specific memory area. For processes that are not time critical, some flexibility can be added, achieving potentially more efficient memory utilization, with dynamic loading from

[1] However, as the degree of multiprogramming increases, so does system overhead (e.g., from I/O queue maintenance); thus, "there is a point at which increased multiprogramming has a negative effect both on memory and on I/O device loadings" [2, p. 486].

backing store. TACFIRE and TSQ-73 employ memory management schemes that are basically similar to static partitioned allocation; TOS is more oriented to paging. The TACFIRE system requires four MCMUs (totalling about 1/2 million 32-bit words); TSQ-73 operates with seven 8K memory modules; TOS has allocated to it four 256K drums (one for programs and three for data files).

d. Language Features Supporting Protection. The language features required for the implementation of a partitioned allocation policy can be derived fairly readily. In any multiprogramming environment, protection is essential for each process's memory space; i.e., a process should not be able to arbitrarily reference addresses in other partitions. Language facilities that aid program modularization (e.g., block structure and separate compilation) help meet this goal. Additionally, language restrictions that prevent a process from treating instructions as data or vice versa, and from dynamically changing the contents of an instruction word, greatly facilitate debugging and are almost essential for reliability. Unfortunately, the run-time costs needed to guarantee these restrictions without suitable support in the computer hardware (e.g., subscript bounds checking on each array reference) may be intolerable for efficiency reasons. Thus the language should preferably allow the user the option of having run-time checking code or not.

e. Additional Language Requirements. In the case of static partitions, no specific HOL features are required, except for the protection mechanisms mentioned in the preceding subparagraph. That is, allocation of memory occurs at system generation (compilation) time, and as long as the system is running there is no deallocation. For dynamic partition specification, the language requirements are also fairly modest, since the algorithms involved are simple. The only noteworthy requirements arise in the machine-dependent area, since dynamic partitioning involves loading and perhaps relocating blocks of storage. To carry out these functions, the programs must be able to treat a memory word as a piece of untyped data.

3. Processor Management.

a. Contrast with Non-Tactical Systems. Processor management is concerned with the assignment of processors to processes. Several distinctive features make tactical

executives simpler in some respects, and more complex in others, when compared with non-tactical systems in the processor management area. Since many TDS contain a fixed number of processes, the problems attendant upon dynamic creation/deletion of processes do not arise. However, it is typical for many of the processes to be time-critical and possibly periodic, which can place severe demands on the processor management facilities to ensure that process deadlines are satisfied.

b. General Functions. The basic functions performed in processor management are as follows [2, p. 9]:

(1) Keeping track of processors and the status of the various processes; the executive program that carries out this duty is called the traffic controller.

(2) Deciding which process should be allocated processor time, when, and how much; this is done by the process scheduler.

(3) Allocating the processor to a selected process; this is done by the dispatcher.

(4) Reclaiming the processor when the process relinquishes it (e.g., during I/O), when it terminates or when it exceeds its time limit.

c. The "Process" Concept. Any set of language features which intends to support the programming of processor management functions will revolve critically around the concept of a process and how it is implemented. Unfortunately, it is precisely in this area where the twin tactical requirements of efficiency and reliability clash most heavily. The problem is that in order for the language to ensure that concurrent processes share resources correctly (i.e., with no possibility for race conditions or deadlock), a runtime expense must be incurred which may be intolerable because of real-time constraints. Nonetheless, it is useful to present an approach to process handling in an HOL which does help attain the reliability goal, since (1) hardware advances in the future may relax some of the efficiency constraints, and (2) it is feasible for the decision to be left to the programmer as to whether run-time checking code will be generated.

d. The "Monitor" Concept. As developed by Brinch Hansen [3] and Hoare [21], the "monitor" concept facilitates the high-level specification of processes and their synchronization. It has been implemented in a language called Concurrent PASCAL [8,9] and has been successfully employed in the development of the SOLO Operating System for the PDP-11/45 [10]. Basically, a process can be viewed as a combination of three types of entities: (1) private data, which cannot be used by any other processes; (2) access rights, which control the shared data that the process may read or update; and (3) a sequential program, which operates on the process's private data as well as on shared data to which it has access rights. Implicit in the process concept is the assumption that several processes may be executing concurrently (or, in the absence of more than one processor, that execution may be interleaved between several processes). This implies that shared data must be used in an orderly fashion, so that race conditions (when the result of a computation depends on the order in which two concurrent processes execute) or deadlock (where each of several processes has exclusive access to some resource needed by another process, and needs some resource currently held by another process) cannot occur. The monitor is a language construct that permits the orderly use of shared data. A monitor is a combination of four entities: (1) a set of shared variables to be used by concurrent processes; (2) a set of routines that operate on the shared variables (concurrent processes can operate on shared data only through calls on these routines); (3) access rights to shared data in other monitors; and (4) initialization routines that set up the shared data at monitor generation time.

e. Synchronization. In order for the monitor construct to ensure orderly access to its shared data, it is assumed that for a given monitor, at most one of its routines can be in execution at a given time. Thus, while a process is executing a monitor routine, it has exclusive access to the shared data; other processes attempting to invoke routines in the same monitor will be delayed by a system-maintained short-term scheduler. Additionally, it is useful for a monitor to allow programmer-controlled scheduling (i.e., synchronization) when requests for data or other resources cannot be satisfied immediately. To facilitate this, queues can be maintained as part of the monitor's shared data for

each resource. The monitor routine invoked by a process to acquire a resource may delay the process in such a queue; this process can then be awakened by another process through a monitor routine invoked to produce or free the resource in question.

f. Data Abstraction. The language facility which comes closest to the monitor concept is data abstraction (as found, e.g., in CLU [11] or ALPHARD [12]), in which the distinction betweer a data structure's representation and its behavioral properties is made explicit. The benefits of data abstraction lie in program understandability and maintainability; accessing data through interface routines necessarily localizes the representational dependencies in the program. These benefits are increased when the program involves process handling, since this application area is notorious for difficulty in program reading and testing.

g. Related Issues. Three important issues that relate to processor management in tactical executive programs are interprocess communication, time-critical scheduling, and interrupt handling.

(1) Interprocess communications. Interprocess communication is handled nicely by a monitor facility, since the message buffer can be maintained as the shared data part of a suitably written monitor.

(2) Time-critical scheduling. Time-critical scheduling requires language features that enable the programmer to declare process parameters such as deadline, repetition period, and required CPU time. Actual scheduling algorithms tend to be simple, for reasons of runtime efficiency (e.g., the "Relative Urgency" method described by Serlin [22]) and do not require special language features.

(3) Interrupt handling. The issue of interrupt handling is relevant at two levels. First is the problem of priority hardware interrupts, generally implemented as automatic traps to specific machine locations. To specify the handlers, a programmer needs an assembly-language-like ability to define the contents of absolute machine addresses; depending on the nature of the handler, either assembly language or an HOL augmented with machine-

oriented features may have to be used. A second variety of interrupt behavior is less machine-dependent and is generally referred to as "exception handling". The basic motivation is one of efficiency: the program should not have to perform an excessive amount of run-time checking to detect conditions whose occurrence is relatively infrequent. Instead, when the exceptional condition occurs, a signal should be generated which results in the scheduling of a process to handle the exception. Language facilities that permit this kind of behavior include the PL/I SIGNAL and ON conditions. Experience with these features has been mixed but generally favorable; evaluating PL/I as a language used for writing the MULTICS operating system, Corbato [28] makes the following comments:

> Some of the features like SIGNAL and ON conditions, which have cost us a lot of grief, at least in principle, have been very graceful ways of smoothly and uniformly handling the overflow conditions and the like, which suddenly trap you down into the guts of the supervisor. In previous systems, we have always had the quandary of how to allow the user to supply his own condition handlers in a convenient way. We're not sure that the price isn't perhaps too high, but the mechanism does look good.

4. Input/Output Device Management.

a. General Functions. The management of I/O devices involves the following activities [2, p. 282]:

(1) Keeping track of the status of all devices.

(2) Enforcing an allocation policy - i.e., deciding which process should get a device, for how long, and when. The main techniques for implementing device management policies are:

(a) Dedicated - in which a device (such as a radar) is assigned to a single process.

(b) Shared - in which several processes can share a device (e.g., a disk).

(c) Virtual - in which one physical device is simulated on another (e.g., spooling).

b. Program Components. These activities are handled by programs called the I/O traffic controller, I/O scheduler, and I/O device handlers, respectively [2, p. 301]. The following subparagraphs identify the major functions performed by these programs and describe the resulting language requirements.

(1) I/O Traffic Controller. The I/O traffic controller keeps track of the status of all devices, control units, and channels. The difficulty of this job depends on the ratio of channels to control units, and control units to devices, since the basic functions of the I/O traffic controller are to determine if a path (from channel to control unit to device) is available to service an I/O request, and, if not, to determine when such a path will be free. To perform these functions, the I/O traffic controller generally maintains blocks of information concerning devices, control units, and channels; this information may include status (busy or not), connected control units or channels, and a queue of waiting processes. The monitor concept described above (subparagraph 3d) provides an adequate facility for performing these functions.

(2) I/O Scheduler. The I/O scheduler generally involves simple algorithms to implement some variety of priority scheduling; no special language facilities are required.

(3) I/O Device Handlers. The I/O device handlers typically set up the channel program, handle error conditions, process I/O interrupts, and provide device-dependent scheduling algorithms. The language features required are similar to those described above in subparagraph 3g.

5. Information Management.

a. Types of Information Management. Information has two basic attributes: representation and interpretation. The term "representation" refers to the physical structuring of the data (e.g., whether to compact sparse matrices), and "interpretation" refers to the routines that define the logical properties of the data. It is useful to distinguish among various types of information management, based on the relative importance of the two attributes. Unfortunately, there is no universally agreed upon terminology; we will follow Madnick and Donovan's choices of the terms file system, data management system, and database system [2, p. 399].

(1) File system. A file system is concerned only minimally with representation (e.g., ASCII vs. binary) and has no involvement with the interpretation of the data.

(2) Data management system. A data management system (DMS) performs representation-dependent operations on data but is independent of the interpretation of the data. An example of DMS is IBM's IMS-II [13].

(3) Data base system. A database system (DBS) is concerned with both representation and interpretation of information. Most on-line retrieval systems are examples of DBS.

b. Overview. It is typical to find a combination of file system, DMS, and DBS in a tactical system. In the next several subparagraphs, we discuss issues that are central to these three types of information management in tactical environments, and indicate the implications of these issues in terms of programming language facilities.

c. Reliability. A basic feature of information management is that the data being maintained have a lifetime which exceeds that of the program that created them. This raises problems in connection with reliability, since it is difficult to ensure that the uses of the data will be consistent with what was intended. For example, in many languages it is possible to create a file comprising integer data, and then later to read information from the file as though it were floating-point. One method for preventing this is to enforce type checking at the file level.

d. Modifiability. It is not uncommon in DMS or DBS for the representation of the data to be modified during the system's life cycle. This is especially likely in systems (such as TOS) where the type of information being handled will change with time. To take care of this situation, the language should enforce the hiding of representational details as fully as possible. A data abstraction facility is useful for this purpose.

e. Protection. A severe problem in any type of information-management system (especially in tactical environments) is the protection of data from unauthorized use. There are several levels at which this issue can be approached: e.g., hardware, personnel, and software. At the hardware level, different modes of execution (privileged vs. user) can isolate some of the obviously "dangerous" instructions to known modules, and bounds registers can prevent "wild" memory references. At the personnel level, security checks on individuals interfacing with the system may prevent malicious attempts at unauthorized access to information. Language features can help at the software level: type checking and program modularization facilities (e.g., block structure, data abstraction, separate compilation) are directly relevant.

f. Size and Complexity of Data Base. An issue of importance in a data-base oriented system like TOS is that of dealing with a large, complicated data base. The language implications for this requirement are the same as for MIS and will be dealt with below (paragraph 3.III.5).

Section IV. REQUIREMENTS OF TDS UTILITY PROGRAMS

1. Types of Utility Programs.

There is a wide variety of utility programs necessary in tactical data systems; the following (cited by Martin [1, pp. 66-67]) are typical: system loading programs, restart programs, diagnostics, file reorganization programs, fall-back programs, library tape and file maintenance programs, data generation programs for program testing, simulators, and debugging aids. For convenience of description, we will divide utility programs into four general areas: Automatic Test Equipment, Simulation, Language Processing, and System Construction. The characteristics and programming language requirements of the first three of these areas will be described in paragraphs 3 through 5 below. We omit a formal discussion of System Construction, because no new language requirements would result: the algorithms involved interface primarily with executive programs that handle I/O.

2. TACS^4.

Many of the functions carried out by utility programs are performed by the Army's Tactical Systems Support Software System (TACS^4). As described in [39, p.3]:

> TACS^4 will provide the support software required for development and maintenance of Tactical Systems throughout their life cycle. This support software will be designed to: increase programmer productivity with programmer, debugging and testing aids; facilitate documentation production with automated documentation aids; assist management by providing selected reports on program status and programmer productivity; and provide automated resource estimating and scheduling routines.

The host machine software in TACS^4 includes: a system controller; compilers; program/data library generation and maintenance routines; a compool generator; a macroprocessor; testing support; a linkage editor; a system tape generator; a program patch facility; automated program verification routines; documentation and management aids such as flowcharter, text editor, and statistics gathering routines. TACS^4 target software includes functions to support "test data input, test execution and instrumentation, and test results analysis for system and acceptance testing" [39, pp. 6-7].

3. Automatic Test Equipment (ATE).

a. Sources. The requirements for ATE applications have been summarized and justified in the DATELS Project Master Plan [40] and in Massachusetts Computer Associates' Language Design Study for ATE [14]. The material in this paragraph is based primarily on these two sources.

b. Linearity of Program Flow. One of the most striking features of ATE programs is the linearity of program flow. Typically, testing is divided into strictly sequential phases, which in turn are divided into subphases; data are used generally to select tests rather than to order them. The effect of this expected linearity on an ATE language is twofold. First, language features that facilitate the specification of long chains of decisions are desirable -- decision tables, for example. Second, in view of anticipated long programs, the language should provide for separate compilation and debugging of individual modules.

c. Input/Output. The nature of input/output (called "signals" in ATE) in ATE programs is somewhat distinctive. First, "input/output is frequent with rather little internal processing between input of one signal and output of the next one" [14, p. 3]. In addition, the kind of processing required involves not only the specific input but also the place in the program where the signal is read and perhaps previous results. The signals tend to have fairly simple structure but are associated with a wide variety of physical units. Moreover, it is possible for the "same" signal to have different specification requirements when directed at different ATE.

d. Diversity of Operating Environments. A distinguishing characteristic of ATE is the variety of system contexts in which ATE programs may be expected to operate. Transfer of programs over different computers and over different ATE is not unusual. The debugging environment may be complex, involving simulation of the ATE, the unit under test (UUT), or both. Testing of ATE programs is unique in that an untested program may damage both ATE and UUT. Language facilities that aid validation include standard features useful in any application (e.g., type checking, high-level control structures) as well as features supporting simulation (see paragraph 4 below).

e. Dependence on Executive Programs. ATE programs make heavy use of a variety of executive programs. Examples cited in [40, pp. 2-10] are the need to define time-critical tasks, to synchronize events and measure time delays, to schedule parallel processes and handle interrupts.

f. Human Factors. "Human factors" requirements in ATE are fairly stringent, since the programming will frequently be done by professional engineers with no previous exposure to computer languages. Teachability, writability, readability, and simplicity are thus important language goals.

4. Simulation.

a. Definition. Simulation can be roughly defined [15, p. 349] as "the technique of representing a dynamic system by a model, in order to gain information about the system through experiments with the model". Tactical systems make heavy use of simulation programs in such areas as personnel training, debugging, ATE, and system performance monitoring.

b. General Requirements. Some general requirements for simulation programming are given by Dahl [15, p. 350]. First, the HOL must present a conceptual framework that will aid the programmer in abstracting the essential properties of the system being modeled. Second, a convenient notation must be provided with which dynamic behavior can be described. Third, programming aids must be present, since simulation models (involving a high degree of parallelism) are generally quite difficult to program and debug.

c. Continuous Simulation. There are basically two approaches to simulation: "continuous" and "discrete". With continuous simulation, a digital computer is programmed to model an analog device. Analog functions such as amplifiers and integrators are transformed into an equivalent set of simultaneous differential equations, and the simulation consists of solving these equations numerically. The language features required for this activity are mainly numeric facilities.

d. Discrete Simulation. Discrete simulation involves approximating a given system by a discrete model. Changes that occur continuously in the real system are captured by a sequence of discrete changes (or "events") in the model.

As stated in [15, p. 351]:

> In contrast to the technique of representing the system as a whole by a set of differential equations, the individual events of a discrete model are often specified in great detail. Many discrete event simulation languages are powerful general-purpose algorithmic languages. The user therefore has the opportunity to describe, to any chosen degree of detail, what is actually "going on" in the given system. Procedures for decision-making can be represented in a realistic way.

Typical computations involve keeping track of where individual "items" are at any point in time, moving items from waiting lines to servicing components and vice versa, timing the processing performed and compiling a statistical profile [23, p. 726].

e. Language Requirements for Discrete Simulation. In this subparagraph we describe the main functional requirements for discrete event simulation and indicate the programming language implications [15, pp. 351-352].

(1) Concurrency. In the system being modeled, several events may occur at the same time. The executive must supply process scheduling facilities, so that the programmer can specify this type of behavior.

(2) Size. Practical simulation programs tend to be large both in code and data. To reduce the storage requirements, the language should provide some variety of dynamic memory allocation, especially in view of the dynamic nature of the system: items may enter or leave the system at random times, and storage should be reserved for an item only while it is needed. List processing facilities are necessary in the language.

(3) Interdependence. The complexity of the system being simulated is reflected in complex relationships among program components. To aid the programmer in describing these interdependencies, the language should provide general logic capabilities; in addition, data types to deal with sets and decision tables are useful.

(4) Statistical analysis. Mathematical features are needed in the language to facilitate such computations as the generation of pseudo-random values, and the calculations of means and standard deviations. A graphical facility is useful for the presentation of histograms.

f. Dependence on Executive Programs. It should be pointed out that there is a large similarity between application programs and simulation programs in TDS in terms of the executive facilities which are required. As stated by DeMillo [41, p. 24], "in both...applications, the 'time' dependent manipulation and description of processes which communicate, compete for resources and are scheduled at seemingly random intervals comprises a reasonably large portion of the programming task". Obviously, there are differences between the two areas: in application programs, time is real (not simulated) and events that require attention demand immediate attention; also, processes that occur in application programs are real processes and not abstractions derived (and possibly simplified) from reality. Despite such differences, however, the language facilities derived above (Section III) for executive programs are also needed to support the programming of simulation software.

5. Language Processing.

a. Overview. We view the language processors (i.e., compilers and assemblers) as utility programs because they are necessary for the development of a tactical system but are not part of the operational system. The discussion will focus on the requirements of compilers (i.e., the language features needed to write a compiler) as opposed to assemblers, since the needs of the latter are subsumed under the former.

b. Compiler Functions. It is convenient to view a compiler as performing three basic functions: lexical analysis (in which the source characters are processed), syntactic analysis (in which the syntactic structure is determined), and semantic analysis (in which object code is generated). Whether these functions are interspersed or performed serially, each will typically have a distinctive set of implementation requirements.

(1) Lexical analysis. The basic processing carried out during lexical analysis is character and string manipulation. In conjunction with this activity, a sizable amount of table handling occurs (e.g., looking up identifier names in a symbol table). There is a need to refer to symbolically-named data (e.g., vocabulary elements in a grammar); a data type that allows the explicit enumeration of such names is helpful.

(2) Syntactic analysis. Syntactic analysis involves manipulation of structured objects (frequently trees) and may require some type of linked list facility. Depending on the methods used, either explicit recursion or a program-maintained pushdown store (stack) will be necessary. For table-driven analyzers, it is generally necessary to be able to specify the packing of data structures, so that storage is conserved.

(3) Semantic analysis. The semantic analysis component performs certain manipulations on the intermediate representation of the program (e.g., optimization) and generates the object code. The programs comprising this component tend to be relatively complex, and they manipulate large and interconnected data structures. The production of object code is obviously machine-dependent; some facilities are required in the language so that target machine instructions and data words can be emitted.

Section V. TINMAN/TDS COMPARISON

1. General Comments on Tinman.

The purpose of this section is to compare the language requirements of tactical data systems, as derived above in Section II through IV, with the "needed characteristics" presented in the Tinman. Although it is outside the scope of this effort to give a detailed, item-by-item review of the Tinman (this has been done by Miller [16] and Hoare [17], among others), we offer the following general observations.

a. Programming Methodology. Despite the differences among the types of programming required for tactical systems, there are basic facilities which a language should provide in order to satisfy the needs of any programming application, and in order to promote sound programming methodology. The Tinman concentrates to a large extent on describing and justifying such facilities. We are in agreement with this approach, since it serves to present a solid foundation for a common HOL; in fact, the programming activity is basically the same, independent of the application, and the Tinman goes quite far in the direction of specifying a set of tools which will facilitate that activity.

b. Implementation Difficulties. We should also point out, however, that some of the facilities proposed in the Tinman are relatively new and have not yet been widely implemented (e.g., exception handling [7, Section V.G.7], while other individually desirable features (such as recursion and parallel processing) sometimes have unpleasant interactions which add complexity and inefficiency to the language and to potential implementations. Hoare makes this point fairly strongly [17, pp. 2-3]:

> The project you have undertaken has started well, but it is still a project of supreme technical difficulty . . . . All those engaged in the project must use this difficulty as a constant excuse to simplify, until it hurts and beyond, to remove features, to make restrictions, and above all to avoid setting any challenge to the implementors or users of the language.

c. Requirements of Executive Programs. It should also be pointed out that there is a relatively small amount of attention given in the Tinman to features which are essential in programming tactical executives (e.g., I/O and parallel processing). Sections B10, G6, G7, G8, J3, and J4 in the Tinman provide a description of such features, but do so at too general a level.

2. Tinman/TDS Comparison Table.

a. Table Format. This paragraph summarizes, in Table I, the match between the Tinman and the language requirements derived above for the various functions performed by tactical systems. The rows in the table correspond to individual sections in the Tinman (see Appendix I for the full text). The columns correspond to specific TDS functions described in this report. Three columns pertain to TDS application programs: mathematically-oriented functions, communication, and display processing. The mathematically-oriented functions encompass tracking (paragraph 2.II.1), guidance (2.II.2), threat evaluation (2.II.3), weapons assignment (2.II.4), electronic warfare (2.II.5), and the TACFIRE-oriented functions (2.II.8). Communication and display processing requirements are discussed in paragraphs 2.II.6 and 2.II.7, respectively. Four columns in Table I pertain to TDS executive programs: memory management (paragraph 2.III.2), processor management (2.III.3), I/O device management (2.III.4), and information management (2.III.5). Four columns in Table I pertain to TDS utility programs: ATE (paragraph 2.IV.2), simulation (2.IV.3), language processing (2.IV.4), and system construction (2.IV.1).

b. Table Entries. Each table entry is either "R", "D", or "U", with the following interpretation: "R" means that the given Tinman facility is Required to support the given type of programming, "D" means Desired, and "U" means Unnecessary. We should mention that the distinctions between these categories are somewhat subjective; also, since each Tinman section generally comprises several described features and supplementary explanation, an "R", "D", or "U" rating refers to the main topic of the section and will not necessarily apply to all of the details. In many cases it is possible to trace the entry back to the individual paragraph in which the TDS function is discussed (e.g. when the Tinman section corresponds to a specific feature). More typically, however, the Tinman

sections describe general facilities needed to satisfy various language goals (e.g. reliability, portability) as opposed to performing specific functions; in these cases the table entry reflects the match between the tactical function and the goals relevant to the Tinman section.

d. Comments. In some cases comments are supplied to justify or expand upon a table entry, or to raise a specific point in connection with the Tinman. An asterisk next to the row number indicates that a comment pertaining to that row will follow the table.

Table I.

Tinman/TDS Comparison Table

	Application Programs			Executive Programs				Utility Programs			
	Math-oriented functions	Communications	Display processing	Memory management	Processor management	I/O device management	Information management	Automatic test equipment	Simulation	Language processing	System construction
A. Data and Types											
* 1. Typed Language	R	R	R	R	R	R	R	R	R	R	R
2. Data Types	R	R	R	R	R	D	R	R	R	R	R
* 3. Precision	R	U	U	U	U	U	U	R	R	D	U
* 4. Fixed-Point Numbers	D	D	U	U	U	U	U	D	D	D	U
* 5. Character Data	U	R	R	U	U	R	R	D	R	R	R
6. Arrays	R	R	R	R	R	D	R	R	R	R	D
7. Records	D	D	D	R	R	R	R	R	R	R	D
B. Operations											
1. Assignment and Reference	R	R	R	R	D	D	R	R	R	R	D
* 2. Equivalence	D	D	D	D	D	D	D	D	D	D	D
* 3. Relationals	R	R	R	R	R	R	R	R	R	R	R
* 4. Arithmetic Operations	R	R	R	R	R	R	R	R	R	R	R
5. Truncation and Rounding	R	R	R	R	R	R	R	R	R	R	R
6. Boolean Operations	R	R	R	R	R	R	R	R	R	R	D
* 7. Scalar Operations	U	U	U	U	U	U	U	U	U	U	U
8. Type Conversion	R	R	R	R	R	R	R	R	R	R	R
9. Changes in Numeric Representation	D	D	D	D	D	D	D	D	D	D	D
*10. I/O Operations	D	R	R	D	D	R	R	R	R	R	R
*11. Power Set Operations	R	R	D	D	R	R	D	R	R	R	D
C. Expressions and Parameters											
* 1. Side Effects	R	R	R	R	R	R	R	R	R	R	R
2. Operand Structure	R	D	D	D	D	D	D	R	R	D	D
3. Expressions Permitted	D	D	D	D	D	D	D	D	D	D	D
4. Constant Expressions	D	D	D	D	D	D	D	D	D	D	D
* 5. Consistent Parameter Rules	U	U	U	U	U	U	U	U	U	U	U
6. Type Agreement in Parameters	R	R	R	R	R	R	R	R	R	R	R
7. Formal Parameter Kinds	D	D	D	D	D	D	D	D	D	D	D
* 8. Formal Parameter Specifications	U	U	U	U	U	U	U	U	U	U	U
* 9. Variable Numbers of Parameters	U	U	U	U	U	U	U	U	U	U	U

Table I.

Tinman/TDS Comparison Table

	Application Programs			Executive Programs				Utility Programs			
	Math-oriented functions	Communications	Display processing	Memory management	Processor management	I/O device management	Information management	Automatic test equipment	Simulation	Language processing	System construction
D. Variables, Literals and Constants											
1. Constant Value Identifiers	D	D	D	D	D	D	D	D	D	D	D
2. Numeric Literals	R	R	R	R	R	R	R	R	R	R	R
3. Initial Values of Variables	D	D	D	D	D	D	D	D	D	D	D
4. Numeric Range and Step Size	D	D	D	D	D	D	D	D	D	D	D
5. Variable Types	D	D	D	D	D	D	R	D	D	R	D
6. Pointer Variables	U	U	D	R	U	U	R	D	R	R	D
E. Definition Facilities											
1. User Definitions Possible	D	D	D	D	D	D	R	D	R	R	D
2. Consistent Use of Types	D	D	D	D	D	D	D	D	D	D	D
3. No Default Declarations	R	R	R	R	R	R	R	R	R	R	R
4. Can Extend Existing Operators	D	U	U	U	U	U	U	U	U	U	U
5. Type Definitions	D	D	D	D	D	D	R	D	R	R	D
6. Data Defining Mechanisms	D	D	D	D	D	D	R	D	R	R	D
7. No Free Union or Subset Types	D	U	D	U	U	U	U	D	U	D	U
8. Type Initialization	D	D	D	D	D	D	D	D	D	D	D
F. Scopes and Libraries											
1. Separate Allocation and Access Allowed	D	D	D	D	D	D	D	D	D	D	D
2. Limiting Access Scope	D	D	D	D	D	D	R	D	D	D	D
3. Compile Time Scope Determination	R	R	R	R	R	R	R	R	R	R	R
4. Libraries Available	R	D	D	D	D	D	D	D	D	D	R
5. Library Contents	D	D	D	D	D	D	D	D	D	D	D
6. Libraries and Compools Indistinguishable	D	D	D	D	D	D	D	D	D	D	D
7. Standard Library Definitions	D	D	D	D	D	D	D	D	D	D	D
G. Control Structures											
1. Kinds of Control Structures	R	R	R	R	R	R	R	R	R	R	R
2. The Go To	R	R	R	R	R	R	R	R	R	R	R

Table I.

Tinman/TDS Comparison Table

	Application Programs			Executive Programs				Utility Programs			
	Math-oriented functions	Communications	Display processing	Memory management	Processor management	I/O device management	Information management	Automatic test equipment	Simulation	Language processing	System construction
3. Conditional Control	D	D	D	D	D	D	D	D	D	D	D
4. Iterative Control	D	D	D	D	D	D	D	D	D	D	D
5. Routines	D	U	U	U	U	U	U	U	D	R	U
6. Parallel Processing	R	R	R	D	R	R	D	R	R	U	D
7. Exception Handling	D	D	D	R	R	R	R	R	R	D	D
8. Synchronization and Real Time	R	R	D	D	R	R	D	R	R	U	D
H. Syntax and Comment Conventions											
1. General Characteristics	D	D	D	D	D	D	D	D	D	D	D
2. No Syntax Extensions	D	D	D	D	D	D	D	D	D	D	D
3. Source Character Set	D	D	D	D	D	D	D	D	D	D	D
4. Identifiers and Literals	D	D	D	D	D	D	D	D	D	D	D
5. Lexical Units and Lines	D	D	D	D	D	D	D	D	D	D	D
6. Key Words	D	D	D	D	D	D	D	D	D	D	D
7. Comment Conventions	D	D	D	D	D	D	D	D	D	D	D
8. Unmatched Parentheses	R	R	R	R	R	R	R	R	R	R	R
9. Uniform Referent Notation	D	D	D	D	D	D	D	D	D	D	D
10. Consistency of Meaning	D	D	D	D	D	D	D	D	D	D	D
I. Defaults, Conditional Compilation and Language Restrictions											
1. No Defaults in Program Logic	D	D	D	D	D	D	D	D	D	D	D
2. Object Representation Specifications Optional	D	D	D	D	D	D	D	D	D	D	D
* 3. Compile-Time Variables	D	D	D	D	D	D	D	D	D	D	D
* 4. Conditional Compilation	D	D	D	D	D	D	D	D	D	D	D
5. Simple Base Language	D	D	D	D	D	D	D	D	D	D	D
6. Translator Restrictions	D	D	D	D	D	D	D	D	D	D	D
7. Object Machine Restrictions	D	D	D	D	D	D	D	D	D	D	D
J. Efficiency Object Representations and Machine Dependencies											
1. Efficiency Object Code	R	R	R	R	R	R	R	D	R	D	D
2. Optimizations Do Not Change Program Effect	R	R	R	R	R	R	R	R	R	R	R

Table I.

Tinman/TDS Comparison Table

	Application Programs			Executive Programs				Utility Programs			
	Math-oriented functions	Communications	Display processing	Memory management	Processor management	I/O device management	Information management	Automatic test equipment	Simulation	Language processing	System construction
3. Machine Language Insertions	D	D	D	R	R	R	R	R	D	R	R
4. Object Representation Specifications	R	R	R	R	R	R	R	D	D	D	D
5. Open and Closed Routine Calls	D	D	D	D	D	D	D	D	D	D	D

Comments on Table I:

A1. In some of the functional areas (communications, executive programs) untyped data are necessary. For example, a memory management routine must be able to deal with typeless storage blocks.

A3. The "R" and "D" entries refer to the need for floating-point data, as opposed to the specific requirement for global (to a scope) specification of precision.

A4. The "D" entries refer to the need for fixed-point data. In tactical applications, binary fixed-point is likely to be more appropriate (for efficiency reasons) than the exact fixed-point facility called for in A4. The "U" entries refer to the fact that the fixed-point facility required by A4 is not necessary. However, some aspects of this facility (e.g., the existence of an INTEGER data type) are needed.

A5. The "R" and "D" entries refer to the need for some kind of character string facility. It is not necessary for character sets to be "treated as any other enumeration type."

B2. This requirement is subject to two possible interpretations:

(1) There is a built-in operation which, for any two (possibly identical) data types T_1 and T_2, will compare for identity any object of type T_1 with any object of type T_2.

(2) There is a built-in operation which, for any data type T, will compare for identity any two objects of type T.

The second interpretation is assumed here.

B3. Relational operators are necessary for numeric data but are not required for arbitrary enumeration types.

B4. A general exponentiation operator is not necessary.

B7. As stated, this requirement is too general. For example, B7 permits a (3x4) array to be added to a (2x3x2) array, and allows a numeric comparison between an array of integers and an integer. See also [16, pp 5-6].

B10. Dynamic (i.e. run-time) assignment and reassignment of I/O devices is not necessary.

B11. The "R" and "D" entries refer to the need for a bitstring data type (which may be realized via a power set).

C1. This characteristic is required for purposes of program transportability.

C5. Despite the intentions of C5, it appears that consistency of rules in the areas cited will increase and not reduce language complexity. For example, parameters to procedures should be capable of being passed by "copy" or by "reference," but it is difficult to apply these concepts to parameters of data types, since the latter will be objects in existence at compile-time.

C8. As explained in [16, p.10], C8 provides a simple macro facility and not the generic procedure capability claimed.

C9. The utility of user-definable procedures with variable numbers of parameters is outweighed by the resulting language complexity.

I3 and I4.
The "D" entries refer to the desirability for some kind of conditional compilation facility (not necessarily the same as that described in I3 and I4).

CHAPTER 3

REQUIREMENTS OF MANAGEMENT INFORMATION SYSTEMS

Section I. AN OVERVIEW OF MANAGEMENT INFORMATION SYSTEMS

1. Army Management Information Systems.

This paragraph describes the objectives of several Management Information Systems for which USACSC is responsible. The description is divided into (a) those systems designed for permanent army installations and (b) those systems designed for use in the field.

a. Systems at Permanent Installations.

(1) Standard Installation/Division Personnel System (SIDPERS). The major functions of SIDPERS are strength accounting, organization and personnel record keeping, interface systems reporting, and management reporting [42]. The subsystems of SIDPERS are the Automated Personnel Requisitioning Subsystem (which generates requisition data for positions projected to be vacant) and the Recommended Assignment Subsystem (which matches personnel qualifications against the requirements of vacant positions).

(2) Standard Army Intermediate Level Supply System (SAILS). As stated in [43], SAILS "is designed to accomplish all stock control, supply management and reporting, and related financial management functions between the CONUS wholesale level and the direct support or separate unit level." SAILS encompasses a basic supply cycle and a variety of supportive subsystems. The basic cycle carries out the following functions: pre-edit of incoming transactions; document history validation; cross-reference file check; stock number finder file/catalog master data file processes; fund code/accounting processing code validation process; balance process (in which the routine supply accounting and recording actions are performed); main document history process; and output preparation, report and statistics process. The supportive subsystems include: weekly financial cycle; weekly demand analysis cycle; monthly stock record support process; monthly file maintenance process; and excess process.

(3) Standard Finance System (STANFINS). As stated in [44], "STANFINS standardizes and automates financial transactions and major operating requirements of installation finance and accounting divisions; creates, updates and maintains base-level financial data banks for retrieval of statistical reports; and produces data required to update higher-echelon data banks ... The current STANFINS applications include appropriation and fund accounting, cost accounting, and ... stock fund and financial inventory accounting. The system provides an audit trail of daily transactions, summarization of daily business, fund availability, cost data, and various reports and automated data for higher headquarters."

(4) Vertical Army Authorization Documents System (VTAADS). As stated in [45], VTAADS "has been developed to support the flow of personnel and materiel authorization data between HQDA and all Army units. The system is designed to create, update and maintain authorization data bases and to interface with asset reporting systems."

b. Systems in the Field.

(1) Combat Service Support System (CS3). This system [46] supports the division supply, maintenance, and personnel functions. The supply functions are carried out by DLOGS-360 (a stock control and inventory management subsystem), a property book management system, and the Army Equipment Status Reporting System. The Maintenance Reporting and Management system encompasses a Maintenance Control System, a Modification Work Order accounting subsystem, and a Materiel Readiness Reporting subsystem. Personnel functions are handled by SIDPERS (Standard Installation/Division Personnel System).

(2) Division Logistics System (DLOGS). This is an interim multicommand automated card processing system [47]. At the division level it manages repair parts, maintains property books, and prepares equipment status reports.

(3) Direct Support Unit/General Support Unit (DSU/GSU). DSU/GSU carries out stock control and inventory accounting functions for direct and general support units [48], using magnetic ledger cards as a storage medium. The programs comprising the system perform conversion from manual to mechanized operation, card editing, updating of Authorized Stockage List (ASL) records, management of fringe item records, cyclic inventory, due-in/due-out reconciliation, asset balance review, and recovery of manual data.

2. General Properties of Management Information Systems.

a. Types of Information Systems. Information systems vary widely in complexity and objectives; in this paragraph we establish a framework for later discussion by summarizing the basic levels at which an information system may function. Following Aron [27], we refer to these levels as "simple data processing system", "information retrieval system", and "information interpretation system".[1]

b. Simple Data Processing System. Such a system comprises "a large number of independent transaction-oriented tasks which summarize inputs to produce reports" [27, p. 217]. While generally involving low implementation and operation costs, the simple data processing system is of benefit more to the clerk than to the manager: "The simple system, being transaction-oriented, sees only single data elements and does not see the interactions between elements; consequently, it lacks the structural information needed to tie tasks together ... it fails to tell management all sides of the story" [27, p. 218].

c. Information Retrieval System. This type of system uses a structured data base which captures the relevant interrelationship among the information. This results in a more flexible and powerful facility than is provided by the simple data processing system, but entails greater costs in development and operation.

d. Information Interpretation System. This kind of system has all the capabilities of both a simple data processing system and an information retrieval system, and it also allows the user "to perform a wide variety of 'scientific' management algorithms which interpret the

[1] Aron uses the term "management information systems" where we use "information interpretation system".

meaning of the data in the system to support decision making" [27, p. 220]. The objectives of such a system include: responsiveness to management's changing needs for information; modifiability in both data and algorithms; convenience of human interface; economy of operation.

e. Management Information System (MIS). We use this term to refer to either an information retrieval or information interpretation system as described above; an MIS may be either batch or on-line in nature. To facilitate the comparison of MIS and TDS in terms of language requirements, we will use the same functional subdivision here as in the previous chapter, viz., application programs, executive programs, and utility programs. The next three sections discuss the requirements for MIS in each of these areas.

Section II. REQUIREMENTS OF MIS APPLICATION PROGRAMS

1. General Attributes.

The specific functions performed by the application programs in a MIS are obviously dependent of the type of system (e.g., inventory control, payroll accounting, etc.), but all share certain common traits. In general, the purpose of an application program is to process a transaction. To do this, it must interpret the command, search a data base to retrieve some information, process this information via an application-specific algorithm, and possibly modify the data base and/or output the results of the transaction.

2. Command Interpretation.

The work involved in interpreting a command can vary considerably in degree of difficulty. In this paragraph, we summarize three representative command language types which illustrate this diversity.

a. Fixed-Format. With this technique (also mentioned above in subparagraph 2.II.7.a) the command which is input is a template that is used to match records in the data base. Command interpretation involves a minimum amount of processing, but the user must be familiar with the representation of the data base in order to compose commands.

b. Procedural. In this case, a command has the form of a program in a simple data-base oriented language. The processing required to interpret a command consists mainly of string handling and table manipulation, and is not unlike the work performed in a programming language interpreter. A procedural command language offers more flexibility than the fixed-format method, but it is less efficient (character handling is generally slow) and still requires the user to be acquainted with the data base structure.

c. Non-Procedural. This is a very general type of command language. The user inputs a high-level description of the transaction (e.g., "FIND SALARIES OF ALL CIVILIAN EMPLOYEES WORKING IN OVERSEAS COMMISSARIES"), and the system in effect transforms this into an equivalent procedural version to carry it out. While potentially the simplest from the user's point of view (no knowledge of the representation of the data base is required), this scheme requires a substantial amount of computation by the

system in order to interpret the command. At the most general level, natural language processing is involved, requiring basically the same types of language facilities for its implementation as are needed in programming language processors.

3. Data Base Accessing.

Issues concerning data base representation and processing requirements for data base accessing will be discussed below in connection with MIS executive programs (Section III). We point out here that a rich file-handling facility and formatted I/O are highly desirable in the language.

4. Computation.

The computations involved in processing a transaction may be either simple or complex, but the language requirements are relatively light. Basic arithmetic and logical facilities are the major need; other features may be required or desired, depending on the specific application.

Section III. REQUIREMENTS OF MIS EXECUTIVE PROGRAMS

1. General Attributes.

a. Main Purpose. The main purpose of MIS executive programs is the same as for TDS executives: viz., to ensure the efficient and reliable sharing of resources among the tasks comprising the system. In this section, we summarize the main similarities and differences between MIS and TDS executives; the organization of this material will parallel the presentation in Section 2.III.

b. Contrast with Tactical Systems. As far as language requirements are concerned, there are two major differences between TDS and MIS executive programs: response time constraints, and data base size. Immediacy of response is critical in tactical environments but is generally not a requirement in MIS: e.g., batch-oriented and/or off-line systems are typical for MIS applications. However, MIS typically must deal with data bases significantly larger than would be found in TDS, which raises problems not usually present in tactical systems. The effect of these differences on the language requirements of the various types of MIS executive programs will be brought out in paragraphs 2 through 5.

2. Memory Management.

Because of the diversity in MIS, a wider variety of memory management techniques are available than for TDS. For simple, batch-oriented environments memory management is not an issue: the "single contiguous allocation" strategy described above (subparagraph 2.III.2.b) is appropriate. At the other extreme, on-line, real-time MIS can use more general techniques. Unlike most tactical systems, MIS may comprise varying numbers of processes which can be dynamically created or deleted (e.g., to service on-line requests from a large number of remote sites): dynamic partitioned allocation may be useful. For these more general memory management systems, the MIS language requirements are basically the same as for TDS.

3. Processor Management.

The issues and language requirements for processor management for MIS are the same as for TDS (paragraph 2.III.3) with the exceptions that dynamic process creation/deletion may be required for some MIS, and time-critical scheduling will usually not be necessary.

4. Input/Output Device Management.

In MIS there are generally a smaller number of I/O devices involved, and the ones used are usually more standard, than in tactical systems. Card readers/punches, printers, terminals, magnetic tapes, and mass storage devices such as disks or drums, are typical in MIS. However, the language features required by programs managing these devices are basically the same as for TDS, as described in paragraph 2.III.4.

5. Information Management.

a. General Properties. In general, a MIS will be a data base system (DBS) in the sense of subparagraph 2.III.5.a; i.e., both representation and interpretation of the data are relevant to the processing programs. The size of the data base and the complexities of the data interrelationships in MIS usually distinguish MIS from tactical systems.

b. Types of Data Base Organizations. The organization of the data base has a direct influence on the performance of a MIS. Following *Stonebraker and Held [24]*, we will consider three types of data base structure which typify most MIS: hierarchical, network, and relational.

(1) Hierarchical. In this type of organization, the data base has a tree structure: each record may have any number of "descendants" but only one "parent", and no loops may appear. This kind of representation facilitates record creation and deletion; if the language allows recursion, data base search procedures may be written fairly straightforwardly. However, the tree organization prohibits a node from being shared by two parents and thus may result in large storage requirements.

(2) Network. This type of structure entails representing the data base as a directed graph (a generalization of a tree), so that the node repetition drawback of the hierarchical approach is circumvented. However, node sharing can make the addition or deletion of records fairly complicated.

(3) Relational. A tree or a network is essentially a binary relation; however, complex data bases may exhibit interconnections which are awkward to capture using only binary relations. Codd

> [25] and others have worked on a general model for representing interconnected data bases, called the "relational" approach, based on the mathematical notion of a relation on an arbitrary number (not just two) of sets. In this model, a relation on sets S1, S2, ..., Sn is represented as a sequence of records, each record comprising elements from S1, S2, ..., Sn, in turn. The flexibility inherent in the relational model is important in achieving a critical goal for DBS: to allow the data base to grow or change while the system is operational.

c. Language Requirements. The language facilities required for developing a DBS of one of the types just outlined are basically a rich data definition facility (such as the one proposed in the Tinman) with programmer control over packing, and features which allow the control of data storage on disk. In addition, data abstraction facilitates the construction of DBS whose dependence on the data representation is encapsulated. It should be pointed out, however, that there are some important differences between DBS and environments where data abstraction is most useful. With data abstraction, it is the set of operations which define the essential properties of the data objects. The situation is somewhat different in DBS, however, as summarized by Hammer [26, p. 59].

> On the other hand, the definition of a shared data base is principally determined by the data which it contains, rather than by one particular set of operations which may be applied to it. Many important uses of a data base do not develop until a long time after its creation. Different coexisting applications may utilize radically different sets of operations. As applications come and go, the abstract view of the data base may undergo major changes, with only the data remaining constant.

Section IV. REQUIREMENTS OF MIS UTILITY PROGRAMS

The utility programs required for MIS are basically the same kinds needed for tactical systems, with the exception that ATE would not be necessary in MIS. As a result, the language requirements for MIS in this area are contained in the TDS language requirements.

Section V. TDS/MIS SUMMARY AND TINMAN/MIS COMPARISON

1. TDS/MIS Summary.

In this paragraph we present a brief review of the main contrasts between the requirements for TDS and MIS.

a. Application Programs. Application programs in tactical systems usually involve mathematical computations with predictable time requirements, which must operate under critical reliability and efficiency constraints. In MIS, there is generally a large diversity of application programs; run-time speed tends to be a goal, but not a stringent requirement.

b. Executive Programs. Tactical executives tend to deal with a fixed number of processes, some of which are time-critical; must service a wide variety of non-standard I/O devices; and interact with a data base which is generally small. In MIS, the number of processes may be fixed but may also vary dynamically; time-criticalness usually does not arise; the number and diversity of I/O devices is usually small; but the size and complexity of the data base can be substantial.

c. Utility Programs. Aside from the absence of ATE in MIS, the requirements for TDS and MIS utility programs are basically the same.

2. Tinman/MIS Comparison

a. Table Format. This paragraph summarizes, in Table II, the match between the Tinman and the language requirements derived for MIS. The rows in the table correspond to individual sections in the Tinman, and the columns correspond to specific MIS functions described in this report. Three columns pertain to MIS application programs: command interpretation (paragraph 3.II.2), data base accessing (3.II.3), and computation (3.II.4). Four columns in in Table II pertain to MIS executive programs: memory management (paragraph 3.III.2), processor management (3.III.3), I/O device management (3.III.4), and information management (3.III.5). Three columns in Table II pertain to MIS utility programs: simulation, language processing, and system construction (section 3.IV).

b. Table Entries. The interpretation of entries is the same as in Table I: "R" means that the given Tinman facility is required to support the given type of programming, "D" means desired, and "U" means unnecessary. It should be observed that there are relatively few differences between Tables I and II: This reflects both the "general purpose" nature of the Tinman and the similarity between TDS and MIS language requirements.

c. Comments. In some cases an added explanation is necessary to justify or expand upon a table entry, or to raise a specific point in connection with the Tinman. An asterisk next to the row number indicates the presence of a comment pertaining to that row. In all cases but one, the comments are identical to those supplied with Table I and will not be repeated in this paragraph. The exception is the comment relating to A4 (Fixed Point Numbers), which follows Table II.

Table II.

Tinman/MIS Comparison Table

	Application Programs			Executive Programs				Utility Programs		
	Command interpretation	Data base accessing	Computation	Memory management	Processor management	I/O device management	Information management	Simulation	Language processing	System construction
A. Data and Types										
* 1. Typed Language	R	R	R	R	R	R	R	R	R	R
2. Data Types	R	R	R	R	R	D	R	R	R	R
* 3. Precision	U	U	D	U	U	U	U	D	U	U
* 4. Fixed Point Numbers	D	U	R	U	U	U	U	D	D	U
* 5. Character Data	R	R	R	U	U	R	R	R	R	R
6. Arrays	R	R	R	R	R	D	R	R	R	D
7. Records	R	R	D	R	R	R	R	R	R	D
B. Operations										
1. Assignment and Reference	R	R	D	R	D	D	R	R	R	D
* 2. Equivalence	D	D	D	D	D	D	D	D	D	D
* 3. Relationals	R	R	D	R	R	R	R	R	R	R
* 4. Arithmetic Operations	R	R	R	R	R	R	R	R	R	R
5. Truncation and Rounding	R	R	R	R	R	R	R	R	R	R
6. Boolean Operations	R	R	D	R	R	R	R	R	R	D
* 7. Scalar Operations	U	U	U	U	U	U	U	U	U	U
8. Type Conversion	R	R	R	R	R	R	R	R	R	R
9. Changes in Numeric Representation	D	D	D	D	D	D	D	D	D	D
*10. I/O Operations	R	R	D	D	D	R	R	R	R	R
*11. Power Set Operations	R	D	D	D	R	R	D	R	R	D
C. Expressions and Parameters										
* 1. Side Effects	R	R	R	R	R	R	R	R	R	R
2. Operand Structure	D	D	D	D	D	D	D	R	D	D
3. Expressions Permitted	D	D	D	D	D	D	D	D	D	D
4. Constant Expressions	D	D	D	D	D	D	D	D	D	D
* 5. Consistent Parameter Rules	U	U	U	U	U	U	U	U	U	U
6. Type Agreement in Parameters	R	R	R	R	R	R	R	R	R	R
7. Formal Parameter Kinds	D	D	D	D	D	D	D	D	D	D
* 8. Formal Parameter Specifications	U	U	U	U	U	U	U	U	U	U
* 9. Variable Number of Parameters	U	U	U	U	U	U	U	U	U	U

Table II.
Tinman/MIS Comparison Table

	Application Programs			Executive Programs				Utility Programs		
	Command interpretation	Data base accessing	Computation	Memory management	Processor management	I/O device management	Information management	Simulation	Language processing	System construction
D. Variables, Literals and Constants										
1. Constant Value Identifier	D	D	D	D	D	D	D	D	D	D
2. Numeric Literals	R	R	R	R	R	R	R	R	R	R
3. Initial Values of Variables	D	D	D	D	D	D	D	D	D	D
4. Numeric Range and Step Size	D	D	D	D	D	D	D	D	D	D
5. Variable Types	R	R	D	D	D	D	R	D	R	D
6. Pointer Variables	R	R	D	R	U	U	R	R	R	D
E. Definition Facilities										
1. User Definitions Possible	R	R	D	D	D	D	R	R	R	D
2. Consistent Use of Types	D	D	D	D	D	D	D	D	D	D
3. No Default Declarations	R	R	R	R	R	R	R	R	R	R
4. Can Extend Existing Operators	U	U	U	U	U	U	U	U	U	U
5. Type Definitions	D	R	D	D	D	D	R	R	R	D
6. Data Defining Mechanisms	R	R	D	D	D	D	R	R	R	D
7. No Free Union or Subset Types	D	U	D	U	U	U	U	U	D	U
8. Type Initialization	D	D	D	D	D	D	D	D	D	D
F. Scopes and Libraries										
1. Separate Allocation and Access Allowed	D	D	D	D	D	D	D	D	D	D
2. Limiting Access Scope	D	R	D	D	D	D	R	D	D	D
3. Compile Time Scope Determination	R	R	R	R	R	R	R	R	R	R
4. Libraries Available	D	D	D	D	D	D	D	D	D	R
5. Library Contents	D	D	D	D	D	D	D	D	D	D
6. Libraries and Compools Indistinguishable	D	D	D	D	D	D	D	D	D	D
7. Standard Library Definitions	D	D	D	D	D	D	D	D	D	D

Table II.
Tinman/MIS Comparison Table

	Application Programs			Executive Programs				Utility Programs		
	Command interpretation	Data base accessing	Computation	Memory management	Processor management	I/O device management	Information management	Simulation	Language processing	System construction
G. Control Structures										
1. Kinds of Control Structures	R	R	R	R	R	R	R	R	R	R
2. The Go To	R	R	R	R	R	R	R	R	R	R
3. Conditional Control	D	D	D	D	D	D	D	D	D	D
4. Iterative Control	D	D	D	D	D	D	D	D	D	D
5. Routines	D	U	U	U	U	U	U	D	R	U
6. Parallel Processing	U	U	U	D	R	R	D	R	U	D
7. Exception Handling	D	D	D	R	R	R	R	R	D	D
8. Synchronization and Real Time	U	D	U	D	D	D	D	R	U	D
H. Syntax and Comment Conventions										
1. General Characteristics	D	D	D	D	D	D	D	D	D	D
2. No Syntax Extensions	D	D	D	D	D	D	D	D	D	D
3. Source Character Set	D	D	D	D	D	D	D	D	D	D
4. Identifiers and Literals	D	D	D	D	D	D	D	D	D	D
5. Lexical Units and Lines	D	D	D	D	D	D	D	D	D	D
6. Key Words	D	D	D	D	D	D	D	D	D	D
7. Comment Conventions	D	D	D	D	D	D	D	D	D	D
8. Unmatched Parentheses	R	R	R	R	R	R	R	R	R	R
9. Uniform Referent Notation	D	D	D	D	D	D	D	D	D	D
10. Consistency of Meaning	D	D	D	D	D	D	D	D	D	D
I. Defaults, Conditional Compilation and Language Restrictions										
1. No Defaults in Program Logic	D	D	D	D	D	D	D	D	D	D
2. Object Representation Specifications Optional	D	D	D	D	D	D	D	D	D	D
* 3. Compile Time Variables	D	D	D	D	D	D	D	D	D	D
* 4. Conditional Compilation	D	D	D	D	D	D	D	D	D	D
5. Simple Base Language	D	D	D	D	D	D	D	D	D	D
6. Translator Restrictions	D	D	D	D	D	D	D	D	D	D
7. Object Machine Restrictions	D	D	D	D	D	D	D	D	D	D

Table II.
Tinman/MIS Comparison Table

	Application Programs			Executive Programs				Utility Programs		
	Command interpretation	Data base accessing	Computation	Memory management	Processor management	I/O device management	Information management	Simulation	Language processing	System construction
J. Efficient Object Representations and Machine Dependencies										
1. Efficient Object Code	D	D	D	R	R	R	R	R	D	D
2. Optimizations Do Not Change Program Effect	R	R	R	R	R	R	R	R	R	R
3. Machine Language Insertions	D	D	D	R	R	R	R	D	R	R
4. Object Representation Specifications	D	D	D	R	R	R	R	D	D	D
5. Open and Closed Routine Calls	D	D	D	D	D	D	D	D	D	D

Comments on Table II

A4. The "D" and "R" entries refer to the need for exact fixed-point data (as described in the Tinman), and not to the binary fixed-point facility appropriate to tactical applications. The "U" entries refer to the fact that the fixed-point facility required by A4 is not necessary. However, some aspects of this facility (e.g., the existence of an INTEGER data type) are needed.

For the other starred entries in Table II, refer to the comments following Table I.

CHAPTER 4
CONCLUSIONS

Section I. CONCLUSIONS ABOUT TDS REQUIREMENTS

1. Similarity to General-Purpose Requirements.

Despite some special requirements for programming tactical systems, it should be emphasized that the basic programming activity is much the same, independent of the application area. The fundamental result of this study is that there is no inconsistency between the basic language requirements for "general purpose" vs. tactical programming. In particular, recent advances in software methodology (and how to achieve them in a language) are especially applicable to tactical systems, in light of the reliability and maintainability objectives. Structured control facilities, and data definition features which separate representation from behavioral properties, are examples of language elements that support this methodology. The Tinman incorporates many other facilities which promote good programming practice.

2. Specific Requirements.

The distinctive needs of tactical systems are concentrated in the requirements for executive programs. Parallel processing, time-critical scheduling, priority interrupt handling, interfacing with operating personnel and non-standard I/O devices: each of these must be carried out in tactical systems, without sacrificing reliability or efficiency. Although the specification of high-level facilities to achieve these goals is still a research area, some recent work in methodology has had some fruitful results (e.g., the monitor concept); we look to these and similar activities in the future to best support the special requirements of tactical systems.

Section II. CONCLUSIONS ABOUT MIS REQUIREMENTS

1. Similarities to General-Purpose Requirements.

The comments in paragraph 4.I.1 above apply to MIS as well as to tactical systems: the programming involved in the development of a MIS is more similar than dissimilar to "general purpose" programming. The facilities described in the Tinman are consistent with the needs of MIS functions.

2. Special Requirements.

The distinctive feature of MIS is the need to operate on large, interconnected data bases; this differentiates MIS from most tactical and general-purpose programs. Recent work on relational data bases has been encouraging in meeting the objective of system flexibility (i.e., allowing the data base to undergo changes while the system is still operating). We look to future work in the data abstraction area for potential language facilities that will support such data bases.

LITERATURE CITED

BOOKS

1. Martin, James. Programming Real-Time Computer Systems. Englewood Cliffs, N.J., Prentice Hall, Inc., 1965.

2. Madnick, Stuart E. and Donovan, John J. Operating Systems. New York, McGraw-Hill Book Company, 1974.

3. Brinch Hansen, Per. Operating System Principles. Englewood Cliffs, N.J., Prentice Hall, Inc., 1973.

4. Genuys, F. (ed.) Programming Languages. London, Academic Press Inc. (London) Ltd., 1968.

REPORTS

5. Dept. of Defense HOL Working Group. Report of Subcommittee on Strawman HOL Requirements, April, 1975.

6. Fisher, David A. "Woodenman" Set of Criteria and Needed Characteristics for a Common DoD High Order Programming Language (working paper), Institute for Defense Analyses, August 1975.

7. Dept. of Defense HOL Working Group. Department of Defense Requirements for High Order Computer Programming Languages "Tinman", March 1976.

8. Brinch Hansen, Per. Concurrent PASCAL Introduction. California Institute of Technology, July 1975.

9. Brinch Hansen, Per. Concurrent PASCAL Report. California Institute of Technology, July 1975.

10. Brinch Hansen, Per. The SOLO Operating System. California Institute of Technology, July 1975.

11. Liskov, Barbara. A Note on CLU. Massachusetts Institute of Technology, Project MAC, Computation Structures Group Memo 112, November 1974.

12. Wulf, William A. Alphard: Toward a Language to Support Structured Programs. Carnegie-Mellon University, Dept. of Computer Science Internal Report, April 1974.

13. Information Management System IMS/360, Application Description Manual (Version 2). Form GH20-0765-1, International Business Machines Corporation, Data Processing Div., White Plains, N.Y., 1971.

14. Massachusetts Computer Associates, Inc. Language Design Study for Automatic Test Equipment. Prepared for Dept. of the Army, Frankford Arsenal, Contract No. DAAA25-72-C-0492, December 1972. (Appears as Section 4, Appendix B, in [40].)

15. Dahl, Ole-Johan. Discrete Event Simulation Languages. (Appears in [4], pp. 349-395.)

16. Miller, James S. Comments on "Tinman" Set of Criteria and Needed Characteristics for a Common DoD High Order Programming Language. Intermetrics, Inc., December 1975.

17. Hoare, C.A.R. Comments on "Tinman" (working paper), December 1975.

18. Goodenough, John B. and Shafer, Lawrence H. A Study of High Level Language Features. SofTech, Inc. Prepared for U.S. Army Electronics Command, Contract No. DAAB07-75-C-0373, February 1976.

JOURNAL/MAGAZINE ARTICLES

19. Aron, J.D. "Real-Time Systems in Perspective". IBM Systems Journal, Vol. 6, No. 1 (1967), pp. 49-67.

20. Chapin, George G. "What Is Different about Tactical Military Operational Programs". National Computer Conference (1973), pp. 787-795.

21. Hoare, C.A.R. "Monitors: An Operating System Structuring Concept". Comm. ACM, Vol. 17, No. 10 (October 1974), pp. 549-557.

22. Serlin, Omri. "Scheduling of Time Critical Processes". Spring Joint Computer Conference (1972), pp. 925-932.

23. Teichroew, Daniel and Lubin, John F. "Computer Simulation - Discussion of the Technique and Comparison of Languages". Comm. ACM, Vol. 9, No. 10 (October 1966), pp. 723-741.

24. Stonebraker, Michael and Held, Gerald. "Networks, Hierarchies, and Relations in Data Base Management Systems". Association for Computing Machinery, Regional Conference, San Francisco (April 1975), pp. 1-9.

25. Codd, E.F. "A Relational Model of Data for Large Shared Data Banks". Comm. ACM, Vol. 13, No. 6 (June 1970), pp. 377-387.

26. Hammer, Michael. "Data Abstractions for Data Bases". Association for Computing Machinery, Proc. of Conference on Data: Abstraction, Definition and Structure (March 1976), pp. 58-59.

27. Aron, J.D. "Information Systems in Perspective". Computing Surveys, Vol. 1, No. 4 (December 1969), pp. 213-236.

28. Corbato, F.J. "PL/I as a Tool for System Programming". Datamation (May 1969).

29. Shaw, Christopher J. "PL/I for C&C? Not Yet". Datamation (December 1968), pp. 26-31.

GOVERNMENT PUBLICATIONS

30. U.S. Army Computer Systems Command. TACFIRE Fact Sheet (Revised). January 1976.

31. U.S. Army Computer Systems Command. TOS Fact Sheet (Revised). January 1976.

32. U.S. Army Computer Systems Command. AN/TSQ-73 Fact Sheet. January 1976.

33. Office of the Project Manager, U.S. Army Tactical Data Systems. Tactical Operations System Operable Segment (TOS^2) System Engineering Study Report. 31 December 1971. (Revision per 28 January 1972).

34. Development Specification for Battalion Operational Computer Program for Air Defense System Guided Missile AN/TSQ-73. Specification No. MI-PC19501, Code Identification 18876. 11 January 1972.

35. Naval Electronics Laboratory Center, San Diego. USN/USMC Future Data Systems Requirements and Their Impact on the All Applications Digital Computer (AADC). Technical Document 239, December 1974.

36. French, Lt. Andrew H. and Mott-Smith, John C. AFSC HOL Standardization Program Phase 1 Report (FY75) (Draft), Hanscom Air Force Base, Bedford, MA, July 1975.

37. Cohen, P.M. The Use of a Communications Oriented Language within a Software Engineering System. Defense Communications Engineering Center, Reston, Va., Technical Note No. 17-75, April 1975.

38. Larson, Maj. Ian W. An Interactive Language Query System for Retrieving Alphanumeric Data from an Army Tactical Data System. Master's Thesis, Faculty of the U.S. Army Command and General Staff College, 1975.

39. U.S. Army Computer Systems Command, Tactical Systems Support Software System ($TACS^4$) System Specifications, Initial Draft. Fort Belvoir, Virginia. 17 March 1975.

40. Dept. of the Army, Defense Focal Point. Defense Automatic Test Equipment Language Standardization, Project Master Plan for Operational Performance Analysis Language System, February 1975.

41. DeMillo, Richard A. Primitives for Tactical Real Time Control Languages Based on SIMULA 67 (Part I: General Language Considerations). Centacs Report No. 50, U.S. Army Electronics Command, Fort Monmouth, N.J., June 1975.

42. U.S. Army Computer Systems Command. SIDPERS Fact Sheet. April 1976.

43. U.S. Army Computer Systems Command. SAILS Fact Sheet. March 1975.

44. U.S. Army Computer Systems Command. STANFINS Fact Sheet. March 1976.

45. U.S. Army Computer Systems Command. VTAADS Fact Sheet. October 1975.

46. U.S. Army Computer Systems Command. CS3 Fact Sheet (Revised). March 1973.

47. U.S. Army Computer Systems Command. DLOGS Fact Sheet. May 1976.

48. U.S. Army Computer Systems Command. DSU/GSU Fact Sheet. May 1976.

Appendix I

Department of Defense

Requirements for High Order

Computer Programming Languages

"Tinman"

June 1976

Section IV

Paragraphs A through J

A. DATA AND TYPES
1. Typed Language
2. Data Types
3. Precision
4. Fixed Point Numbers
5. Character Data
6. Arrays
7. Records

A1. The language will be typed. The type (or mode) of all variables, components of composite data structures, expressions, operations, and parameters will be determinable at compile time and unalterable at run time. The language will require that the type of each variable, and component of composite data structures be explicitly specified in the source programs.

By the type of a data object is meant the set of objects themselves, the essential properties of those objects and the set of operations which give access to and take advantage of those properties. The author of any correct program in any programming language must, of course, know the types of all data and variables used in his programs. If the program is to be maintainable, modifiable and comprehensible by someone other than its author, the the types of variables, operations, and expressions should be easily determined from the source program. Type specifications in programs provide the redundancy necessary to verify automatically that the programmer has adhered to his own type conventions. Static type definitions also provide information at compile time necessary for production of efficient object code. Compile time determination of types does not preclude the inclusion of language structures for dynamic discrimination among alternative record formats (see A7) or among components of a union type (see E6). Where the subtype or record structure cannot be determined until run time, it should still be fully discriminated in the program text so that all the type checks can be completed at compile time.

A2. The language will provide data types for integer, real (floating point and fixed point), Boolean and character and will provide arrays (i.e., composite data structures with indexable components of homogeneous type) and records (i.e., composite data structures with labeled components of heterogeneous type) as type generators.

These are the common data types and type generators of most programming languages and object machines. They are sufficient, when used with a data

definition facility (see E6, D6, and J1), to efficiently mechanize other desired types such as complex or vector.

A3. The source language will require global (to a scope) specification of the precision for floating point arithmetic and will permit precision specification for individual variables. This specification will be interpreted as the maximum precision required by the program logic and the minimum precision to be supported by the object code.

This is a specification of what the program needs, not what the hardware provides. Machine independence, in the use of approximate value numbers (usually with floating point representation), can be achieved only if the user can place constraints on the translator and object machine without forcing a specific mechanization of the arithmetic. Precision specifications, as the minimum supported by the object code, provide all the power and guarantees needed by the programmer without unnecessarily constraining the object machine realization. Precision specifications will not change the type of reals nor the set of applicable operations. Precison specifications apply to arithmetic operations as well as to the data and therefore should be specified once for a designated scope. This permits different precisions to be used in different parts of a program. Specification of the precision will also contribute to the legibility and implementability of programs.

A4. Fixed point numbers will be treated as exact quantities which have a range and a fractional step size which are determined by the user at compile time. Scale factor management will be done by the compiler.

Scaled integers are useful approximations to real numbers when dealing with exact quantity fractional values, when the object machine does not have floating point hardware, and when greater precision is required than is available with the floating point hardware. Integers will also be treated as exact quantities with a step size equal to one.

A5. Character sets will be treated as any other enumeration type.

Like any other data type defined by enumeration (see E6), it should be possible to specify the program literal and order of characters. These properties of the character set would be unalterable at run time. The definition of a character set should reflect on the manner it is used within a program and not necessarily on the

print representation a particular physical device associates with a bit pattern at run time. In general, unless all devices use the same character code, run-time translation between character sets will be required. Widely used character sets, such as, USASCII and EBCDIC will be available in a standard library. Note that access to a linear array filled with the characters of an alphabet, A, and indexed by an alphabet, B, will convert strings of characters from B to A.

A6. The language will require user specification of the number of dimensions, the range of subscript values for each dimension, and the type of each array component. The number of dimensions, the type and the lower subscript bound will be determinable at compile time. The upper subscript bound will be determinable at entry to the array allocation scope.

This is general enough to permit both arrays which can be allocated at compile or load time and arrays which can be allocated at scope entry, but does not permit dynamic change to the size of constructed arrays. It is sufficient to permit allocation of space pools which the user can manage for allocation of more complex data structures including dynamic arrays. The range of subscript values for any given dimension will be a contiguous subsequence of values from an enumeration type (including integers). The preferable lower bound on the subscript range will be the initial element of an enumeration type or zero, because it often contributes to program efficiency and clarity.

A7. The language will permit records to have alternative structures, each of which is fixed at compile time. The name and type of each record component will be specified by the user at compile time.

This provides all that is safe to use in CMS-2 and JOVIAL OVERLAY and in FORTRAN EQUIVALENCE. It permits hierarchically structured data of heterogeneous type, permits records to have alternative structures as long as each structure is fixed at compile time and the choice is fully discriminated at run time, but it does not permit arbitrary references to memory nor the dropping of type checking when handling overlayed structures. The discrimination condition will not be restricted to a field of the record but should be any expression.

B. OPERATIONS

1. Assignment and Reference
2. Equivalence
3. Relationals
4. Arithmetic Operations
5. Truncation and Rounding
6. Boolean Operations
7. Scalar Operations
8. Type Conversion
9. Changes in Numeric Representation
10. I/O Operations
11. Power Set Operations

B1. Assignment and reference operation will be automatically defined for all data types which do not manage their data storage. The assignment operation will permit any value of a given type to be assigned to a variable, array or record component of that type or of a union type containing that type. Reference will retrieve the last assigned value.

The user will be able to declare variables for all data types. Variables are useful only when there are corresponding access and assignment operations. The user will be permitted to define assignment and access operations as part of encapsulated type definitions (see E5). Otherwise, they will be automatically defined for types which do not manage the storage for their data. (See D6 for further discussion).

B2. The source language will have a built-in operation which can be used to compare any two data objects (regardless of type) for identity.

Equivalence is an essential universal operation which should not be subject to restriction on its use. There are many useful equivalence operations for some types and a language definition cannot foresee all these for user defined types. Equivalence meaning logical identity and not bit-by-bit comparison on the internal data representation, however, is required for all data types. Proper semantic interpretation of identity requires that equality and identity be the same for atomic data (i.e., numbers, characters, Boolean values, and types defined by enumeration) and that elements of a disjoint types never be identical. Consequently, its usefulness at run time is restricted to data of the same type or to types with nonempty intersections. For floating point numbers identity will be defined as the same within the specified (minimum) precision.

B3. Relational operations will be automatically defined for numeric data and all types defined by enumeration.

Numbers and types defined by enumeration have an obvious ordering which should be available through relational operations. All six relational operations will be included. It will be possible to inhibit ordering definitions when unordered sets are intended.

B4. The built-in arithmetic operations will include: addition, subtraction, multiplication, division (with a real result), exponentiation, integer division (with integer or fixed point arguments and remainder), and negation.

These are the most widely used numeric operations and are available as hardware operations in most machines. Floating point operations will be precise to at least the specified precision.

B5. Arithmetic and assignment operations on data which are within the range specifications of the program will never truncate the most significant digits of a numeric quantity. Truncation and rounding will always be on the least significant digits and will never be implicit for integers and fixed point numbers. Implicit rounding beyond the specified precision will be allowed for floating point numbers.

These requirements seem obvious, particularly for floating point numbers and yet many of our existing languages truncate the most significant mantissa digits in some mixed and floating point operations.

B6. The built-in Boolean operations will include "and," "or," "not," and "nor." The operations "and" and "or" on scalars will be evaluated in short circuit mode.

Short circuit mode as used here is a semantic rather than an implementation distinction and means that "and" and "or" are in fact control operations which do not evaluate side effects of their second argument if the value of the first argument is "false" or "true," respectively. Short circuit evaluation has no disadvantages over the corresponding computational operations, sometimes produces faster executing code in languages where the user can rely on the short circuit execution, and improves the clarity and maintainability of programs by permitting expressions such

as, "i<=7 & A[i] >x" which could be erroneous were short circuit execution not intended. Note that the equivalence and nonequivalence operations (see B2) are the same as logical equivalence and exclusive-or respectively.

B7. The source language will permit scalar operations and assignment on conformable arrays and will permit data transfers between records or arrays of identical logical structure.

Conformability will require exactly the same number of components (although a scalar can be considered compatible with any array) and one for one compatibility in type. Correspondence will be by position in similarly shaped arrays. In many situations component by component operations are done on array elements. In fact, a primary reason for having arrays is to permit large numbers of similarly treated objects to have a uniform notation. Operations on data aggregates available directly in the source language hide the details of the sequencing and thereby, simplify the program and make more optimizations available. In addition, they permit simultaneous execution on machines with parallel processing hardware. Although component by component operations will be available for built-in composite data structures which are used to define application-oriented structures, that capability will not be automatically inherited by defined data structures. A matrix might be defined using an array, but it will not inherit the array operations automatically. Multiplication for matrices would, for example, be unnatural, confusing and inconvenient if the product operator for matrices were interpreted as a component by component operation instead of cross product of corresponding row and column vectors. Component by component operations also allow operations on character strings represented as vectors of characters and allow efficient Boolean vector operations.

Transfers between arrays or records of identical logical structure are necessary to permit efficient run time conversion from one object representation to another, as might be done when data is packed to reduce peripheral storage requirements and I/O transfer times but need to be unpacked locally to minimize processing costs.

B8. There will be no implicit type conversions but no conversion operation will be required when the type of an actual parameter is a constituent of a union type which is the formal parameter. The language will provide explicit conversions operations among integer, fixed point and floating point data, between the object representation of numbers and their representations as characters, and between fixed point scale factors.

Implicit type conversions which represent changes in the value of data items without an explicit indicator in the program, are not only error prone but can result in unexpected run time overhead.

B9. Explicit conversion operations will not be required between numerical ranges. There will be a run time exception condition when any integer or fixed point value is truncated.

Because ranges do not form closed systems, range validation is not possible at compile time (e.g., "I:=I+1" may be a range error). At best, the compiler might point out likely range errors. (This requirement is optional for hardware installations which do not have overflow detection).

B10. The base language will provide operations allowing programs to interact with files, channels or devices including terminals. These operations will permit sending and receiving both data and control information, will enable programs to dynamically assign and reassign I/O devices, will provide user control for exception conditions, and will not be installation dependent.

Whether the referenced "files" are real or virtual and whether they are hardware devices, I/O channels or logical files depends on the object machine configuration and on the details of its operating system if present. But in any programming system I/O operations ultimately reduce to sending or receiving data and/or control information to a file or to a device controller. These can be made accessible in a HOL in an abstract form through a small set of generic I/O operations (like "read" and "write, " with appropriate device and exception parameters). Note that devices and files are similar in many respects to types, so additional language features may not be required to satisfy this requirement. This requirement, in conjunction with requirement E1, permits user definition of unique equipment and its associated I/O operations as data types within the syntactic and semantic framework provided by the generic operations.

B11. The language will provide operations on data types defined as power sets of enumeration types (see E6). These operations will include union, intersection, difference, complement, and an element predicate.

As with any data type, power sets will be useful only if there are operations which can create, select and interrogate them. Note that this provides only a very

special class of sets but one which is very useful for computations on sets of indicators, flags, and similar devices in monitoring and control applications. More general sets if desired, must be defined using the type definition facilities.

C. EXPRESSIONS AND PARAMETERS

1. Side Effects
2. Operand Structure
3. Expressions Permitted
4. Constant Expressions
5. Consistent Parameter Rules
6. Type Agreement in Parameters
7. Formal Parameter Kinds
8. Formal Parameter Specifications
9. Variable Numbers of Parameters

C1. Side effects which are dependent on the evaluation order among the arguments of an expression will be evaluated left-to-right.

This is a semantic restriction on the evaluation order of arguments to expressions. It provides an explicit rule (i.e., left-to-right) for order of argument evaluation, but allows the implementations to alter the actual order in any way which does not change the effect. This provides the user with a simple rule for determining the effects of interactions among argument evaluations without imposing a strict rule on compilers which are sophisticated enough to detect potential side-effects and optimize through reordering of arguments when the evaluation order does not affect the result. Control operations (e.g., conditional and iterative control structures), of course, must be exceptions to this general rule since control operations are in fact those operations which specify the sequencing and evaluation rules for their arguments.

C2. Which parts of an expression constitute the operands to each operation within that expression should be obvious to the reader. There will be few levels of operator hierarchy and they will be widely recognized.

Care must be taken to ensure that the operator/operand structure of expressions is not psychologically ambiguous (i.e., to guarantee that the parse implemented by the language is the same as intended by the programmer and understood by those reading the program). This kind of problem can be minimized by having few precedence levels and parsing rules by allowing explicit parentheses to specify the intended execution order, and by requiring explicit parentheses when the execution order is of significance to the result within the same precedence level (e.g., "X divided by Y divided by Z" and "X divided by Y multiplied by Z"). The user will not be able to define new operator precedence rules nor change the precedence of existing operators.

C3. Expressions of a given type will be permitted anywhere in source programs where both constants and references to variables of that type are allowed.

This is an example of not imposing arbitrary restrictions and special case rules on the user of the source language. Special mention is made here only because so many languages do restrict the form of expressions. FORTRAN, for example, has a list of seven different syntactic forms for subscript expressions, instead of allowing all forms of arithmetic expressions.

C4. Constant expressions will be allowed in programs anywhere constants are allowed, and constant expressions will be evaluated before run time.

The ability to write constant expressions in programs has proven valuable in languages with this capability, particularly with regard to program readability and in avoiding programmer error in externally evaluating and transcribing constant expressions. They are most often used in declarations. There is no need, however, that constant expressions impose run time costs for their evaluation. They can be evaluated once at compile time or if this is inconvenient because of incompatibilities between the host and object machines, the compiler can generate code for their evaluation at load time. In any case, the resulting value should be the same (at least within the stated precision) regardless of the object machine (see D2). Allowing constant expressions in place of constants can improve the clarity, correctness and maintainability of programs and does not impose any run time costs.

C5. There will be a consistent set of rules applicable to all parameters, whether they be for procedures, for types for exception handing, for parallel processes, for declarations, or for built-in operators. There will be no special operations (e.g., array substructuring) applicable only to parameters. Uniformity and consistency contributes to ease of learning,

implementing and using a language; allows the user to concentrate on the programming task instead of the language; and leads to more readable, understandable, and predictable programs.

C6. Formal and actual parameters will always agree in type. The number of dimensions for array parameters will be determinable at compile time. The size and subscript range for array parameters need not be determinable at compile time, but can be passed as part of the parameter.

Type transfers hidden in procedure calls with incompatible formal and actual parameters whether intentional or accidental have long been a source of program errors and of programs which are difficult to maintain. On the other hand, there is no reason why the subscript ranges for arrays cannot be passed as part of the arguments. Some notations permit such parameters to be implicit on the call side. Formal parameters of a union type will be considered conformable to actual parameters of any of the component types.

C7. There will be only four classes of formal parameters. For data there will be those which act as constants representing the actual parameter value at the time of call, and those which rename the actual parameter which must be a variable. In addition, there will be a formal parameter class for specifying the control action when exception conditions occur and a class for procedure parameters.

The first class of data parameter acts as a constant within the procedure body and cannot be assigned to nor changed during the procedures execution; its corresponding actual parameter may be any legal expression of the desired type and will be evaluated once at the time of call. The second class of data parameter renames the actual parameter which must be a variable, the address of the actual parameter variable will be determined by (or at) the time of call and unalterable during execution of the procedure, and assignment (or reference) to the formal parameter name will assign (or access) the variable which is the actual parameter. These are the only two widely used parameter passing mechanisms for data and the many alternatives (at least 10 have been suggested) add complexity and cost to a language without sufficiently increasing the clarity or power. A language with exception handling capability must have a way to pass control and related data through procedure call interfaces. Exception handling control parameters will be specified on the call side only when needed. Actual procedure parameters will be restricted to those of similar (explicit or implicit) specification parts.

C8. Specification of the type, range, precision, dimension, scale and format of parameters will be optional in the procedure declaration. None of them will be alterable at run time.

Optional formal parameter specification permits the writing of generic procedures which are instantiated at compile time by the characteristics of their actual parameters. It eliminates the need for compile time "type" parameters. This generic procedure capability, for example, allows the definition of stacks and queues and their associated operations on data of any given type without knowing the data type when the operations are defined.

C9. There will be provision for variable numbers of arguments, but in such cases all but a constant number of them must be of the same type. Whether a routine can have a variable number of arguments must be determinable from its description and the number of arguments for any call will be determinable at compile time.

There are many useful purposes for procedures with variable numbers of arguments. These include intrinsic functions such as "print," generalizations of operations which are both commutative and associative such as "max" and "min," and repetitive application of the same binary operation such as the Lisp "list" operation. The use of variable number of argument operations need not and will not cause relaxation of any compile time checks, require use of multiple entry procedures allow the number of actual parameters to vary at run time, nor require special calling mechanisms. If the parameters which can vary are limited to a program specified type treated as any other argument on the call side and as elements of an array within the procedure definition, full type checking can be done at compile time. There will be not prohibition on writing a special case of a procedure for a particular number of arguments.

D. VARIABLES, LITERALS AND CONSTANTS

1. Constant Value Identifiers
2. Numeric Literals
3. Initial Values of Variables
4. Numeric Range and Step Size
5. Variable Types
6. Pointer Variables

D1. The user will have the ability to associate constant values of any type with identifiers.

The use of identifiers to represent constant values has often made programs more readable, more easily modifiable and less prone to error when the value of a constant is changed. Associating constant values with an identifier is preferable to assigning the value to a variable because it is then clearly marked in the program as a constant, can be automatically checked for unintentional changes, and often can have a more efficient object representation.

D2. The language will provide a syntax and a consistent interpretation for constants of built-in data types. Numeric constants will have the same value (within the specified precision) in both programs and data (input or output).

Literals are needed for all atomic data types and should be provided as part of the language definition for built-in types. Regardless of the source of the data and regardless of the object machine the value of constants should be the same. For integers it should be exact and for reals it should be the same within the specified precision. Compiler writers, however, would disagree. They object to this requirement on two grounds: that it is too costly if the host and object machines are different and that it is unnecessary if they are the same. In fact, all costs are at compile time and must be insignificant compared to the life time costs resulting from object cope containing the wrong constant values. As for being unnecessary, there have been all too many cases of different values from program and data literals on the same machine because the compile time and run time conversion packages were different and imprecise.

D3. The language will permit the user to specify the initial values of individual variables as part of their declaration. Such variables will be initialized at the time of their apparent allocation (i.e., at entry to allocation scope). There will be no default initial values.

The ability to initialize variables at the time of their allocation will contribute to program clarity, but a requirement to do so would be an arbitrary and sometimes costly decision to the user. Default initial values on the other hand, contribute to neither program clarity nor correctness and can be even more costly at run time. It is usually a programming error if a variable is accessed before it is initialized. It is desirable that the translator give a warning when a path between the declaration and use of a variable omits initialization. Whether a variable will be assigned is in general an unsolvable problem, but it is sometimes determinable whether assignments occur on potential paths. In the case of arrays, it it possible at compile time only to determine that some components (but not necessarily which) have been initialized. There will be provision (at user option) for run time testing for initialization.

D4. The source language will require its users to specify individually the range of all numeric variables and the step size for fixed point variables. The range specifications will be interpreted as the maximal specifications will be interpreted as the maximal range of values which will be assigned to a variable and the minimal range which must be supported by the object code. Range and step size specifications will not be interpreted as defining new types.

Range specifications are a special form of assertion. They aid in understanding and determining the correctness of programs. They can also be used as additional information by the compiler in deciding what storage and allocation to use (e.g., half words might be more efficient for integers in the range 0 to 1000). Range specifications also offer the opportunity for the translator to insert range tests automatically for run time or debug time validation of the program logic. With the ranges of variables specified in the program, it becomes possible to perform many subscript bounds checks at compile time. These bounds, checks, however, can be only as valid as the range specifications which cannot in general be validated at compile time. Range specifications on approximate valued variables (usually with floating point implemetation) also offer the possibility of their implementation using fixed point hardware.

D5. The range of values which can be associated with a variable, array, or record component will be any built-in type, any defined type or a contiguous subsequence of any enumeration type.

There should not be any arbitrary restrictions on the structure of data. This permits arrays to be components of records or arrays and permits records to be components of arrays.

D6. The language will provide a pointer mechanism which can be used to build data with shared and/or recursive substructure. The pointer property will only affect the use of variables (including array and record components) of some data types. Pointer variables will be as safe in their use as are any other variables.

Assignment to a pointer variable will mean that the variable's name is to act as an additional label (or reference) on the datum being assigned. Assignment to a nonpointer variable will mean that the variable's name it to label a copy of the object being assigned. For data without alterable component structure or alterable component values, there is no functional difference between reference to multiple copies and multiple references to a single copy. Consequently, pointer/nonpointer will be a property only of variables for composite types and of composite array and record components Because the pointer/nonpointer property applies to all variables of a given type, it will be specified as part of the type definition. The use of pointers will be kept safe by prohibiting pointers to data structures whose allocation scope is narrower than that of the pointer variable.

Such a restriction is easily enforced at compile time using hierarchical scope rules providing there is no way to dynamically create new instances of the data type. In the latter case, the dynamically created data can be allocated with full safety using a (user or library defined) space pool which is either local (i.e., own) or global to the type definition. If variables of a type do not have the pointer property then dynamic storage allocation would be required for assignment unless their size is constant and known at the time of variable allocation. Thus, the nonpointer property will be permitted only for types (a) whose data have a structure and size which is constant in the type definition or (b) which manage the storage for their data as part of the type definition. Because pointers are often less expensive at run time than nonpointers and are subject to fewer restrictions, the specification of the nonpointer property will be explicit in programs (this is similar to the Algol-60 issue concerning the explicit specification of "value" (i.e., nonpointer) and "name" (i.e. pointer). The need for pointers is obvious in building data structures with shared or recursive substructures; such as, directed graphs, stacks, queues, and list structures. Providing pointers as absolute address types, however, produces gaps in the type checking and scope mechanisms. Type and access restricted pointers will provide the power of general pointers without their undesirable characteristics.

E. DEFINITION FACILITIES
1. User Definitions Possible
2. Consistent Use of Types
3. No Default Declarations
4. Can Extend Existing Operators
5. Type Definitions
6. Data Defining Mechanisms
7. No Free Union or Subset Types
8. Type Initialization

E1. The user of the language will be able to define new data types and operations within programs.

The number of specialized capabilities needed for a common language is large and diverse. In many cases, there is no consensus as to the form these capabilities should take in a programming language. The operational requirements dictating specific specialized language capabilities are volatile and future needs cannot always be foreseen. No language can make available all the features useful to the broad spectrum of military applications, anticipate future applications and requirements or even provide a universally "best" capability in support of a single application area. A common language needs capability for growth. It should contain all the power necessary to satisfy all the applications and the ability to specialize that power to the particular application task. A language with defining facilities for data and operations often makes it possible to add new application-oriented structures and to use new programming techniques and mechanisms through descriptions written entirely within the language. Definitions will have the appearance and costs of features which are built into the language while actually being catalogued accessible application packages. The operation definition facility will include the ability to define new infix operators (but see H2 for restrictions). No programming language can be all things to all people, but a language with data and operation definition facilities can be adapted to meet changing requirements in a variety of areas.

The ability to define data and operations is well within the state of the art. Operation definition facilities in the form of subroutines have been available in all general purpose programming languages since at least the time of early FORTRANs. Data definition facilities have been available in a variety of programming languages for almost 10 years and reached their peak with a large number of extensible languages(Stephen A. Schuman (Ed.) Proceedings of the International Symposium on Extensible Languages, SIGPLAN Notices, Vol. 6, No. 12, December 1971. Also, C. Christensen and C.J. Shaw (Ed.), Proceedings of the Extensible Language Symposium, SIGPLAN Notices 4, August 1969.) (over 30) in 1968 and shortly thereafter. A trend toward more abstract and less machine-oriented data

specification mechanisms has appeared more recently in PASCAL(Niklaus Wirth, "An Assessment of the Programming Language PASCAL, "Proceedings of the International Conference on Reliable Software 21-23 April 1973, p. 23-30). Data type definitions, with operations and data defined together, are used in several languages including SIMULA-67(Jacob Palme, "SIMULA as a Tool for Extensible Program Products, "SIGPLAN Notices, Vol. 9, No. 4, February 1974). On the other hand, there is currently much ferment as to what is the proper function and form of data type definitions.

E2. The "use" of defined types will be indistinguishable from built-in types.

Whether a type is built-in or defined within the base will not be determinable from its syntactic and semantic properties. There will be no ad hoc special cases nor inconsistent rules to interfere with and complicate learning, using and implementing the language. If built-in features and user defined data structures and operations are treated in the same way throughout the language so that the base language, standard application libraries and application programs are treated in a uniform manner by the user and by the translator, then these distinctions will grow dim to everyone's advantage. When the language contains all the essential power, when few can tell the difference between the base language and library definitions, and when the introduction of new data types and routines does not impact the compiler and the language standards, then there is little incentive to proliferate languages. Similarly, if typed definitions are processed entirely at compile time and the language allows full program specification of the internal representation, there need be no penalty in run time efficiency for using defined types.

E3. Each program component will be defined in the base language, in a library, or in the program. There will be no default declarations.

As programmers, we should not expect the translator to write our programs for us (at least in the immediate future). If we somehow know that the translator's default convention is compatible with our needs for the case at hand we should still document the choice so others can understand and maintain our programs. Neither should we be able to delay definitions (possibly forget them) until they cause trouble in the operational system. This is a special case of requirement I1.

E4. The user will be able, within the source language, to extend existing operators to new data types.

When an operation is an abstraction of an existing operation for a new type or is a generalization of an existing operation, it is inconvenient, confusing and misleading to use any but the existing operator symbol or function named. The translator will not assume that commutativity of bulit-in operations is preserved by extensions, and any assumptions about the associativity of built-in or extended operations will be ignored by the translator when explicit parentheses are provided in an expression.

E5. Type definitions in the source language will permit definition of both the class of data objects comprising the type and the set of operations applicable to that class. A defined type will not automatically inherit the operations of the data with which it is represented.

Types define abstract data objects with special properties. The data objects are given a representation in terms of existing data structures, but they are of little value until operations are available to take advantage of their special properties. When one obtains access to a type, he needs its operations as well as its data. Numeric data is needed in many applications but is of little value to any without arithmetic operations. The definable operations will include constructors, selectors, predicates, and type conversions.

E6. The data objects comprising a defined type will be definable by enumeration of their literal names, as Cartesian products of existing types (i.e., as array and record classes), by discriminated union (i.e., as the union of disjoint types) and as the power set of an enumeration type. These definitions will be processed entirely at compile time.

The above list comprises a currently known set of useful definitional mechanisms for data types which do not require run time support, as do garbage collection and dynamic storage allocation. In conjunction with pointers (see D6), they provide many of the mechanisms necessary to define recursive data structures and efficient sparse data structures.

E7. Type definitions by free union (i.e., union of non-disjoint types) and subsetting are not desired.

Free union adds no new power not provided by discriminated union, but does require giving up the security of types in return for programmer freedom. Range and

subset specifications on variables are useful documentation and debugging aids, but will not be construed as types. Subsets do not introduce new properties or operations not available to the superset and often do not form a closed system under the superset operations. Unlike types, membership in subsets can be determined only at run time.

E8. When defining a type, the user will be able to specify the initialization and finalization procedures for the type and the actions to be taken at the time of allocation and deallocation of variables of that type.

It is often necessary to do bookkeeping or to take other special action when variables of a given type are allocated or deallocated. The language will not limit the class of definable types by withholding the ability to define those actions. Initialization might take place once when the type is allocated (i.e., in its allocation scope) and would be used to set up the procedures and initialize the variables which are local to the type definition. These operations will be definable in the encapsulation housing the rest of the type definition.

F. SCOPES AND LIBRARIES

1. Separate Allocation and Access Allowed
2. Limiting Access Scope
3. Compile Time Scope Determination
4. Libraries Available
5. Library Contents
6. Libraries and Compools Indistinguishable
7. Standard Library Definitions

F1. The language will allow the user to distinguish between scope of allocation and scope of access.

The scope of allocation or lifetime of a program structure is that region of the program for which the object representation of the structure should be present. The allocation scope defines the program scope for which own variables of the structure must be maintained and identifies the time for initialization of the structure. The access scope defines the regions of the program in which the allocated structure is accessible to the program and will never be wider than the allocation scope. In some cases the user may desire that each use of a defined program structure be independent (i.e., the allocation and accessing scopes would be identical). In other cases, the various accessing scopes might share a common allocation of the structure.

F2. The ability to limit the access to separately defined structures will be available both where the structure is defined and where it is used. It will be possible to associate new local names with separately defined program components.

Limited access specified in a type definition is necessary to guarantee that changes to data representations and to management routines which purportedly do not affect the calling programs are in fact safe. By rigorously controlling the set of operations applicable to a defined type, the type definition guarantees that no external use of the type can accidentally or intentionally use hidden nonessential properties of the type. Renaming separately defined programming components is necessary to avoid naming conflicts when they are used.

Limited access on the call side provides a high degree of safety and eliminates nonessential naming conflicts without limiting the degree of accessibility which can be built into programs. The alternative notion, that all declarations which are external to a program segment should have the same scope, is inconvenient and

costly in creating large systems which are composed from many subsystems because it forces global access scopes and the attendant naming conflicts on subsystems not using the defined items.

F3. The scope of identifiers will be wholly determined at compile time.

Identifiers will be declared at the beginning of their scope and multiple use of variable names will not be allowed in the same scope. Except as otherwise explicitly specified in programs, access scopes will be lexically embedded with the most local definition applying when the same identifier appears at several levels. The language will use the above lexically embedded scope rules for declarations and other definitions of identifiers to make them easy to recognize and to avoid errors and ambiguities from multiple use of identifiers in a single scope.

F4. A variety of application-oriented data and operations will be available in libraries and easily accessible in the language.

A simple base alone is not sufficient for a common language. Even though in theory such a language provides the necessary power and the capability for specialization to particular applications, the users of the language cannot be expected to develop and support common libraries under individual projects There will be broad support for libraries common to users of well recognized application areas. Application libraries will be developed as early as possible.

F5. Program components not defined within the current program and not in the base language will be maintained in compile time accessible libraries. The libraries will be capable of holding anything definable in the language and will not exclude routines whose bodies are written in other source languages.

The usefulness of a language derives primarily from the existence and accessibility of specialized application-oriented data and operations. Whether a library should contain source or object code is a question of implementation efficiency and should not be specified in the definition of the source language, but the source language description will always be available. It should be remembered, however, that interfaces cannot be validated at program assembly time without some equivalent of their source language interface specifications, that object modules are machine-dependent and, therefore, not portable, that source code is often more compact than object code, and that compilers for simple languages can sometimes

compile faster than a loader can load from relocatable object programs. Library routines written on other languages will not be prohibited provided the foreign routine has object code compatible with the calling mechanisms used in the Common HOL and providing sufficient header information (e.g., parameter types, form and number) is given with the routine in Common HOL form to permit the required compile time checks at the interface.

F6. Libraries and Compools will be indistinguishable. They will be capable of holding anything definable in the language, and it will be possible to associate them with any level of programming activity from systems through projects to individual programs. There will be many specialized compools or libraries any user specified subset of which is immediately accessible from a given program.

Compools have proven very useful in organizing and controlling shared data structures and shared routines. A similar mechanism will be available to manage and control access to related library definitions.

F7. The source language will contain standard machine independent interfaces to machine dependent capabilities, including peripheral equipment and special hardware.

The convenience, ease of use and savings in production and maintenance costs resulting from using high order languages come from being able to use specialized capabilities without building them from scratch. Thus, it is essential that high level capabilities be supplied with the language. The idea is not to provide all the many special cases in the language, but to provide a few general cases which will cover the special cases.

There is currently little agreement on standard operating system, I/O, or file system interfaces. This does not preclude support of one or more forms for the near term. For the present the important thing is that one be chosen and made available as a standard supported library definition which the user can use with confidence.

G. CONTROL STRUCTURES

1. Kinds of Control Structures
2. The Go To
3. Conditional Control
4. Iterative Control
5. Routines
6. Parallel Processing
7. Exception Handling
8. Synchronization and Real Time

G1. The language will provide structured control mechanisms for sequential, conditional, iterative, and recursive control. It will also provide control structures for (pseudo) parallel processing, exception handling and asynchronous interrupt handling.

These mechanisms, hopefully, provide a spanning set of control structures. The most appropriate operations in several of these areas is an open question. For the present, the choice will be a spanning set of composable control primitives each of which is easily mapped onto object machines and which does not impose run time charges when it is not used. Whether parallel processing is real (i.e., by multiprocessing) or is synthesized on a single sequential processor, is determined by the object machine, but if programs are written as if there is true parallel processing (and no assumption about the relative speeds of the processors) then the same results will be obtained independent of the object environment.

It is desirable that the number of primitive control structures in the language be minimized, not by reducing the power of the language, but by selecting a small set of composable primitives which can be used to easily build other desired control mechanisms within programs. This means that the capabilities of control mechanisms must be separable so that the user need not pay either program clarity or implementation costs for undesired specialized capabilities. By these criteria, the Algol-60 "FOR" would be undesirable because it imposes the use of a loop control variable, requires that there be a single terminal condition and that the condition be tested before each iteration. Consequently, "FOR" cannot be composed to build other useful iterative control structures (e.g., FORTRAN "DO"). The ability to compose control structures does not imply an ability to define new control operations and such an ability to define new control operations, and such an ability is in conflict with the limited parameter passing mechanisms of C7.

G2. The source language will provide a "GO TO" operation applicable to program labels within its most local scope of definition.

The "GO TO" is a machine level capability which is still needed to fill in any gaps which might remain in the choice of structured control primitives, to provide compatibility for translitterating programs written in older languages, and because of the wide familiarity of current practitioners with its use. The language should not, however, impose unnecessary costs for its presence. The "GO TO" will be limited to explicitly specified program labels at the same scope level. Neither should the language provide specialized facilities which encourage its use in dangerous and confusing ways. Switches, designational expressions, label variables, label parameters and numeric labels are not desired. Switches here refer to the unrestricted switches which are generalizations of the "GO TO" and not to case statements which are a general form for conditionals(see G3). This requirements should not be interpreted to conflict with the specialized form of control transfer provided by the exception handling control structure of G7.

G3. The conditional control structures will be fully partitioned and will permit selection among alternative computations based on the value of a Boolean expression, on the subtype of a value from a discriminated union, or on a computed choice among labeled alternatives.

The conditional control operations will be fully partitioned (e.g., an "ELSE" clause must follow each "IF THEN") so the choice is clear and explicit in each case. There will be some general form of conditional which allows an arbitrary computation to determine the selected situation (e.g., Zahn's device(Donald E. Knuth, "Structured Programming with go to Statements," ACM Computer Surveys, Vol. 6, No. 4, December 1974) provides a good solution to the general problem). Special mechanisms are also needed for the more common cases of the Boolean expression (e.g., "IF THEN ELSE") and for value or type discrimination (e.g., "CASE" on one of a set of values or subtype of a union).

G4. The iterative control structure will permit the termination condition to appear anywhere in the loop, will require control variables t be local to the iterative control, will allow entry only at the head of the loop, and will not impose excessive overhead in clarity or run the execution costs for common special case termination conditions (e.g., fixed number of iterations or elements of an array exhausted).

In its most general form, a programmed loop is executed repetitively until some computed predicate becomes true. There may be more than one terminating predicate, and they might appear anywhere in the loop. Specialized control structures (e.g., "WHILE DO") have been used for the common situation in which the

AD-A038 214 INTERMETRICS INC CAMBRIDGE MASS F/G 9/2
LANGUAGE REQUIREMENTS REPORT.(U)
JUL 76 B M BROSGOL, J L FELTY DAHC26-76-C-0006
UNCLASSIFIED IR-179-2 USACSC-AT-76-06 NL

2 OF 2
AD
A038214

END

DATE
FILMED
5-77

termination conditions precedes each iteration. The most common case is termination after a fixed number of iterations and a specialized control structure should be provided for that purpose (e.g., FORTRAN "DO" or Algol-60 "FOR"). A problem which arises in many programming languages is that loop control variables are global to the iterative control and thus, will have a value after loop termination, but that value is usually an accident of the implementation. Specifying the meaning of control variables after loop termination in the language definition resolves the ambiguity but must be an arbitrary decision which will not aide program clarity or correctness, and may interfere with the generation of efficient object code. Loop control variables are by definition variables used to control the repetitive execution of a programmed loop and as such will be local to the loop body, but at loop termination it will be possible to pass their value (or any other computed value) out of the loop, conveniently and efficiently.

G5. Recursive as well as nonrecursive routines will be available in the source language. It will not be possible to define procedures within the body of a recursive procedure.

Recursion is desirable in many applications because it contributes directly to their elegance and clairty and simplifies proof procedures. Indirectly, it contributes to the reliability and maintainability of some programs. Recursion is required in order to avoid unnecessarily opaque, complex and confusing programs when operating on recursive data structures. Recursion has not been widely used in DoD software because many programming languages do not provide recursion, practitioners are not familiar with its use, and users fear that its run time costs are to high. Of these, only the run time costs would justify its exclusion from the language.

A major run time cost often attributed to recursion is the need for the presence of a set of "display" registers which are used to keep track of the addresses of the various levels of lexically imbedded environments and which must be managed and updated at run time. The display, however, is necessary only in programs in which routines access variables which are global to their own definition, but local to a more global recursive procedure. This possibility can easily be removed by prohibiting the definition of procedures within the body of a recursive procedure. The utility of such a combination of capabilities is very questionable, and this single restriction will eliminate all added execution costs for nonrecursive procedures in programs which contain recursive procedures.

As with any other facility of the language, routines should be implemented in the most efficient manner consistent with their use and the language should be designed so that efficient implementations are possible. In particular, the most possible regardless of whether the language or even the program contains recursive

procedures. When any routine makes a procedure call as its last operation before exit (and this is quite common for recursive routines) the implementation might use the same data area for both routines, and do a jump to the head of the called procedure thereby saving much of the overhead of a procedure call and eliminating a return. The choice between recursive and nonrecursive routines involves trade-offs. Recursive routines can aid program clarity when operating on recursive data, but can detract from clarity when operating on iterative data. They can increase execution time when procedure call overhead is greater than loop overhead and can decrease execution times when loop overhead is the more expensive. Finally, program storage for recursive routines is often only a small fraction of that for a corresponding iterative procedure, but the data storage requirements are often much larger because of the simultaneous presence of several activations of the same procedure.

G6. The source language will provide a parallel processing capability. This capability should include the ability to create and terminate (possibly pseudo) parallel processes and for these processes to gain exclusive use of resources during specified portions of their execution.

A parallel processing capability is essential in embedded computer applications. Programs must send data to, receive data from, and control many devices which are operating in parallel. Multiprogramming (a form of pseudo paralell processing) is necessary so that many programs within a system can meet their differing real time constraints. The parallel processing capability will minimally provide the ability to define and call parallel processing and the ability to gain exclusive use of system resources in the form of data structures, devices and pseudo devices. This latter ability satisfies one of the two needs for synchronization of parallel processes. The other is required in conjunction with real time constraints (see G8).

The parallel processing capability will be defined as true parallel (as opposed to coroutine) primitives, but with the understanding that in most implementations the object computer will have fewer processors (usually one) than the number of parallel paths specified in a program. Interleaved execution in the implementation may be required.

The parallel processing features of the language should be selected to eliminate any unnecessary overhead associated with their use. The costs of parallel processes are primarily in run time storage management. As with recursive routines most accessing and storage management problems can be eliminated by prohibiting complex interactions with other language facilities where the combination has little if any utility. In particular, it will not be possible to define a parallel routine within the

body of a recursive routine and it will not be possible to define any routine including parallel routines within the body of those parallel routines which can have multiple simultaneous activations. If the language permits several simultaneous activations of a given parallel process then it might require the user to give a upper bound on the number which can exist simultaneously. The latter requirement is reasonable for parallel processes because it is information known by the programmer and necessary to the maintainer, because parallel processes cannot normally be stacked, and because it is necessary for the compilation of efficient programs.

G7. The exception handing control structure will permit the user to cause transfer of control and data for any error or exception situation which might occur in a program.

It is essential in many aplications that there be no program halts beyond the user's control. The user must be able to specify the action to be taken on any exception situation which might occur within his program. The exception handling mechanism will be parameterized so data can be passed to the recovery point. Exception situations might include arithmetic overflow, exhaustion of available space, hardware errors, any user defined exceptions and any run time detected programming error.

The user will be able to write programs which can get out of an arbitrary nest of control and intercept it at any embedding level desired. The exception handling mechanism will permit the user to specify the action to be taken upon the occurrence of a designated exception within any given access scope of the program. The transfers of control will, at the users option, be either forward in the program (but never to a narrower scope of access or out of a procedure) or out of the current procedure through its dynamic (i.e., calling structure. The latter form requires an exception handling formal parameter class (see C7).

G8. There will be source language features which permit delay on any control path until some specified time or situation has occurred, which permit specification of the relative priorities among parallel control paths, which give access to real time clocks, which permit asynchronous hardware interrupts to be treated as any other exception situation.

When parallel or pseudo parallel paths appear in a program it must be possible to specify their relative priorities and to synchronize their executions. Synchronization can be done either through exclusive access to data (see G6) or through delays terminated by designated situations occurring within the program.

These situations should include the elapse of program specified time intervals, occurrence of hardware interrupts and those designated in the program. There will be no implicit evaluation of program determined situations. Time delays will be program specifiable for both real and simulated times.

H. SYNTAX AND COMMENT CONVENTIONS

1. General Characteristics
2. No Syntax Extensions
3. Source Character Set
4. Identifiers and Literals
5. Lexical Units and Lines
6. Key Words
7. Comment Conventions
8. Unmatched Parentheses
9. Uniform Referent Notation
10. Consistency of Meaning

H1. The source language will be free format with an explicit statement delimiter, will allow the use of mnemonically significant identifiers, will be based on conventional forms, will have a simple uniform and easily parsed grammar, will not provide unique notations for special cases, will not permit abbreviation of identifiers or key words, and will be syntactically unambiguous.

Clarity and readability of programs will be the primary criteria for selecting a syntax. Each of the above points can contribute to program clarity. The use of free format, mnemonic identifiers and conventional forms allows the programmer to use notations which have their familiar meanings to put down his ideas and intentions in the order and form that humans think about them, and to transfer skills he already has to the solution of the problem at hand. A simple uniform language reduces the number of cases which must be dealt with by anyone using the language. If programs are difficult for the translator to parse they will be difficult for people. Similar things should use the same notations with the special case processing reserved for the translator and object machine. The purpose of mnemonic identifiers and key words is to be informative and increase the distance between lexical units of programs. This does not prevent the use of short identifiers and short key words.

H2. The user will not be able to modify the source language syntax. Specifically, he will not be able to modify operator hierarchies, introduce new precedence rules, define new key word forms or define new infix operator precedences.

If the user can change the syntax of the language, then he can change the basic character and understanding of the language. The distinction between

semantic extensions and syntactic extensions is similar to that between being able to coin new words in English or being able to move to another natural language. Coining words requires learning those new meanings before they can be used, but at the same time increases the power the language for some application areas. Changing the grammar, (e.g., Franglais, the use of French grammar with interspersed English words) however, undermines the basic understanding of the language itself, changes the mode of expression, and removes the commonalities which obtain between various specializations of the language. Growth of a language through definition of new data and operations and the introduction of new words and symbols to identify them is desirable, but there should be no provision for changing the grammatical rules of the language. This requirement does not conflict with E4 and does not preclude associating new meanings with existing infix operators.

H3. The syntax of source language programs will be composable from a character set suitable for publication purposes, but no feature of the language will be inaccessible using the 64 character ASCII subset.

A common language should use notations and a character set convenient for communicating algorithms, programs, and programming techniques among its users. On the other hand, the language should not require special equipment (e.g., card readers and printers) for its use. The use of the 64 character ASCII subset will make the language compatible with the federal information processing standard 64 character set, FIPS-1, which has been adopted by the U.S.A. Standard code for Information Interchange (USASCII). The language definition will specify the translation from the publication language into the restricted character set.

H4. The language definition will provide the formation rules for identifiers and literals. These will include literals for numbers and character strings and a break character for use internal to identifiers and literals.

Lexical units of the language should be defined in a simple uniform and easily understood manner. Some possible break characters are the space(W. Dijkstra, coding examples in Chapter I, "Notes in Structured Programming," in Structured Programming by O.-J. Dahl, E. W. Dijkstra and C.A.R. Moore, Academic Press, 1972. & Thomas A. Standish, "A Structured Program to Play Tic-Tac-Toe," notes for Information and Computer Science 3 course at Univ. of California-Irvine, October 1974) (i.e., any number of spaces or end-of-line), the underline and the tilde. The space cannot be used if identifiers and user defined infix operators are lexically indistinguishable, but in such a case the formal grammar for the language would be ambiguous (see H1). A literal break character contributes to the readability of

programs and makes the entry of long literals less error prone. With a space as a break character one can enter multipart (i.e., more than one lexical unit) identifiers such as "REAL TIME CLOCK" or long literals, such as, "3.14159 26535 89793." Use of a break can also be used to guarantee that missing quote brackets on character literals do not cause errors which propagate beyond the net end-of-line. the language should require separate quoting of each line of a long literal: "This is a long" "literal string".

H5. There will be no continuation of lexical units across lines, but there will be a way to include object characters such as end-of-line in literal strings.

Many elementary input errors arise at the end of lines. Programs are input on line oriented media but the concept of end-of-line is foreign to free format text. Most of the error prone aspects of end-of-line can be eliminated by not allowing lexical units to continue over lines. The sometimes undesirable effects of this restriction can be avoided by permitting identifiers and literals to be composed from more than one lexical unit (see H4) and by evaluating constant expressions at compile time (see C4).

H6. Key words will be reserved, will be very few in number, will be informative, and will not be usable in contexts where an identifier can be used.

By key words of the language are meant those symbols and strings which have special meaning in the syntax of programs. They introduce special syntactic forms such as are used for control structures and declarations or the are used as infix operators, or as some form of parenthesis. Key words will be reserved, that is unusable as identifiers, to avoid confusion and ambiguity. Key words will be few in number because each new key word introduces another case in the parsing rules and thereby adds to complexity in understanding the language, and because large numbers of key words inconvenience and complicate the programmer's task of choosing informative identifiers. Key words should be concise, but being information is more important than being short. A major exception is the key word introducing a comment; it is the comment and not its key word which should do the informing. Finally, there will be no place in a source language program in which a key word can be used in place of an identifier. That is, functional form operations and special data items built into the language or accessible as a standard extension will not be treated as key words but will be treated as any other identifier.

H7. The source language will have a single uniform comment convention. Comments will be easily distinguishable from code, will be introduced by a single (or possibly two) language defined characters, will permit any combination of characters to appear, will be able to appear anywhere reasonable in programs, will automatically terminate at end-of-line if not otherwise terminated, and will not prohibit automatic reformatting of programs.

These are all obvious points which will encourage the use of comments in programs and avoid their error prone features in some existing languages. Comments anywhere reasonable in a program will not be taken to mean that they can appear internal to a lexical unit, such as, an identifier, key word, or between the opening and closing brackets of a character string. One comment convention which nearly meets these criteria is to have a special quote character which begins comments and with either the quote or an end-of-line ending each comment. This allows both embedded and line-oriented comments.

H8. The language will not permit unmatched parentheses of any kind.

Some programming languages permit closing parentheses to be omitted. If, for example, a program contained more "BEGINs" than "ENDs" the translator might insert enough "ENDs" at the end of the program to make up the difference. This makes programs easier to write because it sometimes saves writing several "ENDs" at the end of programs and because it eliminates all syntax errors for missing "ENDs." Failure to require proper parentheses matching makes it more difficult to write correct programs. Good programming practice requires that matching parentheses be included in programs whether or not they are required by the language. Unfortunately, if they are not required by the language then there can be no syntax check to discover where errors were made. The language will require full parentheses matching. This does not preclude syntactic features such as "case x of s1, s2...sn end case" in which "end" is paired with a key word other than "begin." Nor does it alone prohibit open forms such as "if-then-else-."

H9. There will be a uniform referent notation.

The distinction between function calls and data reference is one of representation, not of use. Thus, there will be no language imposed syntactic distinction between function calls and data selection. If, for example, a computed function is replaced by a lookup table there should be no need to change the calling program. This does not preclude the inclusion of more than one referent notation.

H10. No language defined symbols appearing in the same context will have essentially different meanings.

This contributes to the clarity and uniformity of programs, protects against psychological ambiguity and avoids some error prone features of extant languages. In particular, this would exclude the use of = to imply both assignment and equality, would exclude conventions implying that parenthesized parameters have special semantics (as with PL/1 subroutines), and would exclude the use of an assignment operator for other than assignment (e.g., left hand side function calls). It would not, however, require different operator symbols for integer, real or even matrix arithmetic since these are in fact special cases of the same abstract operations and would allow the use of generic functions applicable to several data types.

I. DEFAULTS, CONDITIONAL COMPILATION AND LANGUAGE RESTRICTIONS

1. No Defaults in Program Logic
2. Object Representation Specifications Optional
3. Compile Time Variables
4. Conditional Compilation
5. Simple Base Language
6. Translator Restrictions
7. Object Machine Restrictions

I1. There will be no defaults in programs which affect the program logic. That is, decisions which affect program logic will be made either irrevocably when the language is defined or explicitly in each program.

The only alternative is implementation dependent defaults with the translator determining the meaning of programs. What a program does, should be determinable from the program and the defining documentation for the programming language. This does not require that binding of all program properties be local to each use. Quite the contrary, it would, for example, allow automatic definition of assignment for all variables or global specification of precision. What it does require is that each decision be explicit: in the language definition, global to some scope, or local to each use. Omission of any selection which affects the program logic will be treated as an error by the translator.

I2. Defaults will be provided for special capabilities affecting only object representation and other properties which the programmer does not know or care about. Such defaults will always mean that the programmer does not care which choice is made. The programmer will be able to override these defaults when necessary.

The language should be oriented to provide a high degree of management control and visibility to programs and toward self-documenting programs with the programmer required to make his decisions explicit. On the other hand, the programmer should not be forced to overspecify his programs and thereby cloud their logic, unnecessarily eliminate opportunities for optimization, and misrepresent arbitrary choices as essential to the program logic. Defaults will be allowed, in fact, encouraged in don't care situations. Such defaults will include data representations (see J4), open vs. closed subroutine calls (see J5), and reentrant vs. nonreentrant code generation.

I3. The user will be able to associate compile time variables with programs. These will include variables which specify the object computer model and other aspects of the object machine configuration.

When a language has different host and object machines and when its compilers can produce code for several configurations of a given machine, the programmer should be able to specify the intended object machine configuration. The user should have control over the compile time variables used in his program. Typically they would be associated with the object computer model, the memory size, special hardware options, the operating system if present, peripheral equipment or other aspects of the object machine configuration. Compile time variables will be set outside the program, but available for interrogation within the program (see I4 and C4).

I4. The source language will permit the use of conditional statements (e.g., case statements) dependent on the object environment and other compile time variables. In such cases the conditional will be evaluated at compile time and only the selected path will be compiled.

An environmental inquiry capability permits the writing of common programs and procedures which are specialized at compile time by the translator as a function of the intended object machine configuration or of other compile time variables (see I3). This requirement is a special case of evaluation of constant expressions at compile time (see C4). It provides a general purpose capability for conditional compilation.

I5. The source language will contain a simple clearly identifiable base or kernel which houses all the power of the language. To the extent possible, the base will be minimal with each feature providing a single unique capablity not otherwise duplicated in the base. The choice of the base will not detract from the efficiency, safety, or understandability of the language.

The capabilities available in any language can be partitioned into two groups, those which are definable within the base and those which provide an essential primitive capability of the language. The smaller and simpler the base the easier the language will be to learn and use. A clearly delineated base with features not in the base defined in terms of the base, will improve the ease and efficiency of learning, implementing and maintaining the language. Only the base need be implemented to make the full source language capability available.

Base features will provide relatively low level general purpose capabilities not yet specialized for particular applications. There will be no prohibition on a translator incorporating speciaized optimizations for particular extensions. Any extension provided by a translator will, however, be definable within the base language using the built-in definition facilities. Thus, programs using the extension will be translatable by any compiler for the language but not necessarily with the same object efficiency.

16. Language restrictions which are dependent only on the translator and not on the object machine will be specified explicitly in the language definition.

Limits on the number of array dimensions, the length of identifiers, the number of nested parentheses levels in expressions, or the number of identifiers in programs are determined by the translator and not by the object machine. Ideally, the limits should be set so high that no program (save the most abrasive) encounters the limits. In each case, however: (a) some limit must be set, (b) whatever the limit, it will impose on some and therefore must be known by the users of the translator, (c) letting each translator set its own limits means that programs will not be portable, (d) setting the limits very high requires that the translator be hosted only on large machines and (e) quite low limits do not impose significantly on either the power of the language or the readability of programs. Thus, the limits should be set as part of the language definition. They should be small enough that they do not dominate the compiler and large enough that they do not interfere with the usefulness of the language. If they were set at say the 99 percent level based on statistics from existing DoD computer programs the limits might be a few hundred for numbers of identifiers and less than ten in the other cases mentioned above.

17. Language restrictions which are inherently dependent only on the object environment will not be built into the language definition or any translator.

Limits on the amount of run time storage, access to specialized peripheral equipments, use of special hardware capabilities and access to real time clocks are dependent on the object machine and configuration. The translator will report when a program exceeds the resources or capabilities of the intended object machine but will not build in arbitrary limits of its own.

J. EFFICIENT OBJECT REPRESENTATIONS AND MACHINE DEPENDENCIES

1. Efficient Object Code
2. Optimizations Do Not Change Program Effect
3. Machine Language Insertions
4. Object Representation Specifications
5. Open and Closed Routine Calls

J1. The language and its translators will not impose run time costs for unneeded or unused generality. They will be capable of producing efficient code for all programs.

The base language and library definitions might contain features and capabilities which are not needed by everyone, or at least, not be everyone all the time. The language should not force programs to require greater generality than they need. When a program does not use a feature or capability it should pay no run time cost for the feature being in the language or library. When the full generality of a feature is not used, only the necessary (reduced) cost should be paid. Where possible, language features (such as, automatic and dynamic array allocation, process scheduling, file management and I/O buffering) which require run time support packages should be provided as standard library definitions and not as part of the base language . The user will not have to pay time and space for support packages he does not use. Neither will there be automatic movement of programs or data between main store and backing shore which is not under program control (unless the object machine has virtual memory with underlying management beyond the control of all its users). Language features will result in special efficient object codes when their full generality is not used. A large number of special cases should compile efficiently. For example, a program doing numerical calculations on unsubscripted real variables should produce code no worse than FORTRAN. Parameter passing for single argument routines might be implemented much less expensively than multiple argument routines.

One way to reduce costs for unneeded capabilities is to have a base language whose data structures and operations provide a single capability which is composable and has a straight-forward implementation in the object code of conventional architecture machines. If the base language components are easily composable they can be used to construct the specialized structures needed by specific applications, if they are simple and provide a single capability they will not force the use of unneeded capabilities in order to obtain needed capabilities, and if they are compatible with the features normally found in sequential uniprocessor digital computers with random access memory they will have near minimum or at least low cost implementation on many object machines.

J2. Any optimizations performed by the translator will not change the effect of the program.

More simply, the translator cannot give up program reliability and correctness, regardless of the excuse. Note that for most programming languages there are few known safe optimizations and many unsafe ones. The number of applicable safe optimizations can be increased by making more information available to the compiler and by choosing language constructs which allow safe optimizations. This requirement allows optimization by code motion providing that motion does not change the effect of the program.

J3. The source language will provide encapsulated access to machine dependent hardware facilities including machine language code insertions.

It is difficult to be enthusiastic about machine language insertions. They defeat the purpose of machine independence constrain the implementation techniques complicate the diagnostics, impair the safety of type checking, and detract from the reliability, readability, and modifiability of programs. The use of machine language insertions is particularly dangerous in multiprogramming applications because they impair the ability to exclude, "a priori," a large class of time-dependent bugs. Rigid enforcement of scope rules by the compiler in real time applications is a powerful tool to ensure that one sequential process will not interfere with others in an uncontrolled fashion. Similarly, when several independent programs are executed in an interleaved fashion, the correct execution of each may depend on the others not having improperly used machine language insertions.

Unfortunately machine language insertions are necessary for interfacing special purpose devices, for accessing special purpose hardware capabilities, and for certain code optimizations on time critical paths. Here we have an example of Dijkstra's dilemma in which the mismatch between high level language programming and the underlying hardware is unacceptable and there is no feasible way to reject the hardware. The only remaining alternative is to "continue bit pushing in the old way, with all the known ill effects." Those ill effects can, however, be constrained to the smallest possible perimeter in practice if not in theory. The ability to enter machine language should not be used as an excuse to exclude otherwise needed facilities from the HOL; the abstract description of programs in the HOL should not require the use of machine language insertions. The semantics of machine language insertions will be determinable from the HOL definition and the object machine description alone and not dependent on the translator characteristics Machine language insertions will be encapsulated so they can be easily recognized and so that it is clear which variables and program identifiers are accessed within the insertion. The machine language insertions will be permitted only within the body of compile

time conditional statements (see I4) which depend on the object machine configuration (see I3). They will not be allowed interspersed with executable statements of the source language.

J4. It will be possible within the source language to specify the object presentation of composite data structures. These descriptions will be optional and encapsulated and will be distinct from the logical description. The user will be able to specify the time/space trade-off to the translator. If not specified, the object representation will be optimal as determined by the translator.

It is often necessary to give detailed specifications of the object data representations to obtain maximum density for large data files to meet format requirements imposed by the hardware of peripheral equipment, to allow special optimizations on time critical paths, or to ensure compatibility when transferring data between machines.

It will be possible to specify the order of the fields, the width of fields, the presence of don't care fields, and the position of word boundaries. It will be possible to associate source language identifiers (data or program) with special machine addresses. The use of machine dependent characteristics of the object representation will be restricted as with machine dependent code (see J3). When multiple fields per word are specified the compiler may have to generate some form of shift and mask operations for source program references and assignments to those variables (i.e., fields). As with machine-language insertions, object data specifications should be used sparingly and the language features for their use must be Spartan, nor grandiose.

If the object representation of a composite data object is not specified in the source program, there will be no specific default guaranteed by the translator. The translator might, for example, attempt to minimize access time and/or memory space in determining the object representation. It might, depending on the object machine characteristics, assign variables and fields of records to full words, but assign array elements to the smallest of bits, bytes, half words, words or exact multiple words permitted by the logical description.

J5. The programmer will be able to specify whether calls on a routine are to have an open or closed implementation. An open and a closed routine of the same description will have identical semantics.

The use of inline open procedures can reduce the run time execution costs significantly in some cases. There are the obvious advantages in eliminating the parameter passing, in avoiding the saving of return marks, and in not having to pass arguments to and from the routine. A less obvious, but often more important advantage in saving run time costs is the ability to execute constant portions of routines at compile time and, thereby, eliminate time and space for those portions of the procedure body at run time. Open routine capability is especially important for machine language insertions.

The distinction between open and closed implementation of a routine is an efficiency consideration and should not affect the function of the routine. Thus, an open routine will differ from a syntax macro in that (a) its global environment is that of its definition and not that of its call and (b) multiple occurrences of a formal value (i.e., read only) parameter in the body have the same value. If a routine is not specified as either open or closed the choice will be optimal as determined by the translator.